Heavy-Light Decomposition

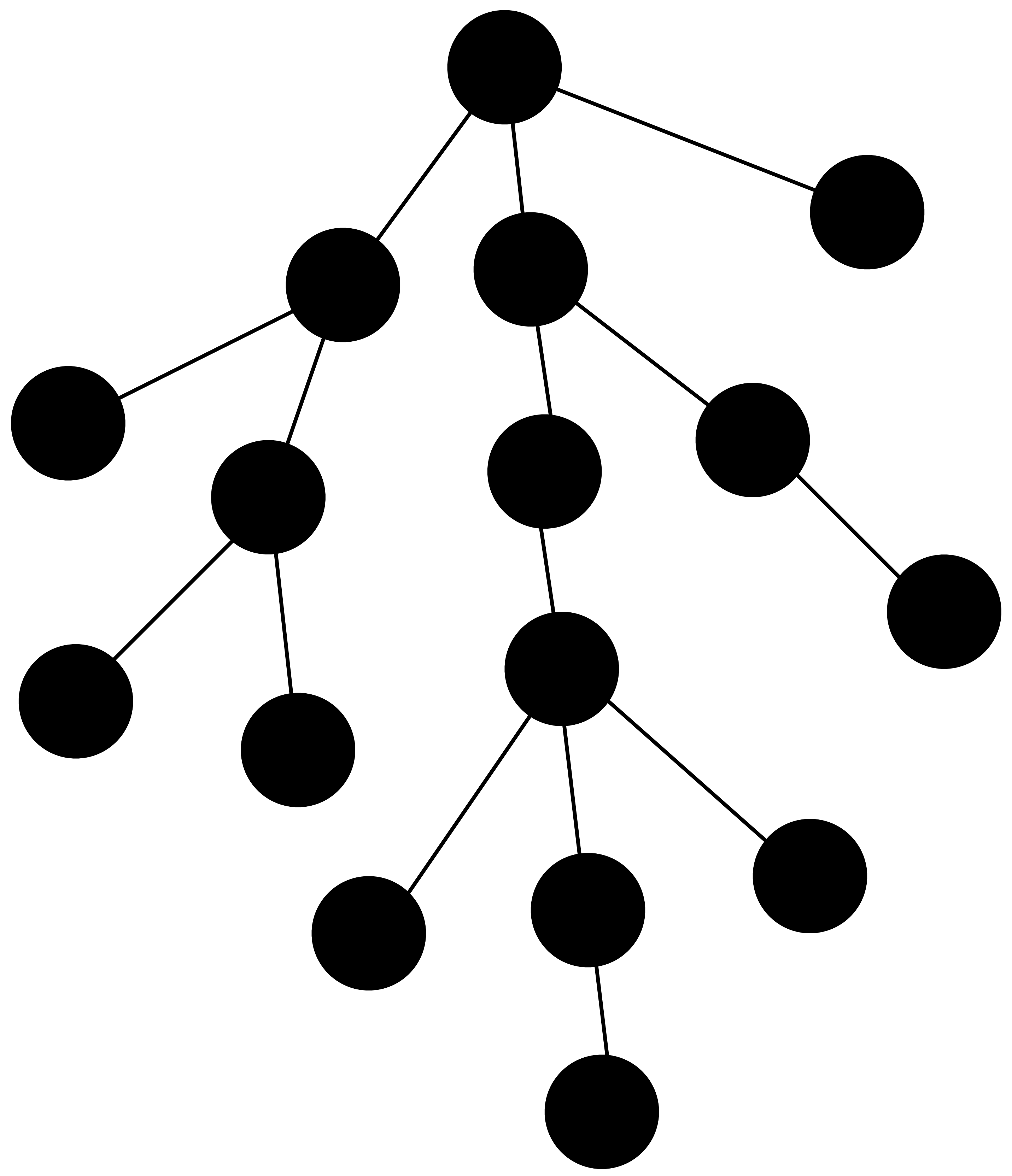
IOI Training Camp 2 - 2024

Noah Jacobsen

CSES 1137 - Subtree Queries

Given a tree consisting of N nodes, answer Q queries of the form:

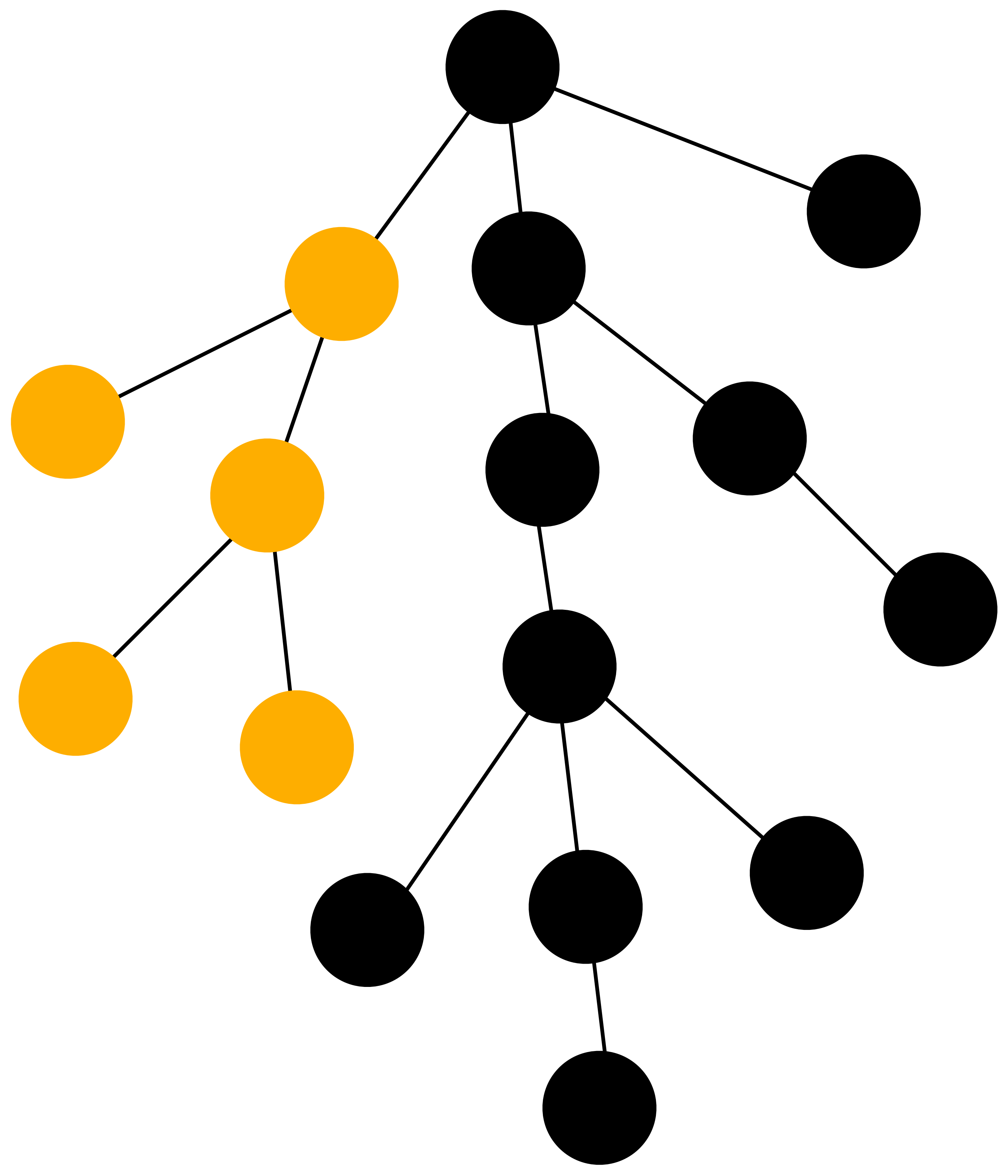
1. Change the value of node s to v
2. Find the sum of all nodes in the subtree of s



CSES 1137 - Subtree Queries

Given a tree consisting of N nodes, answer Q queries of the form:

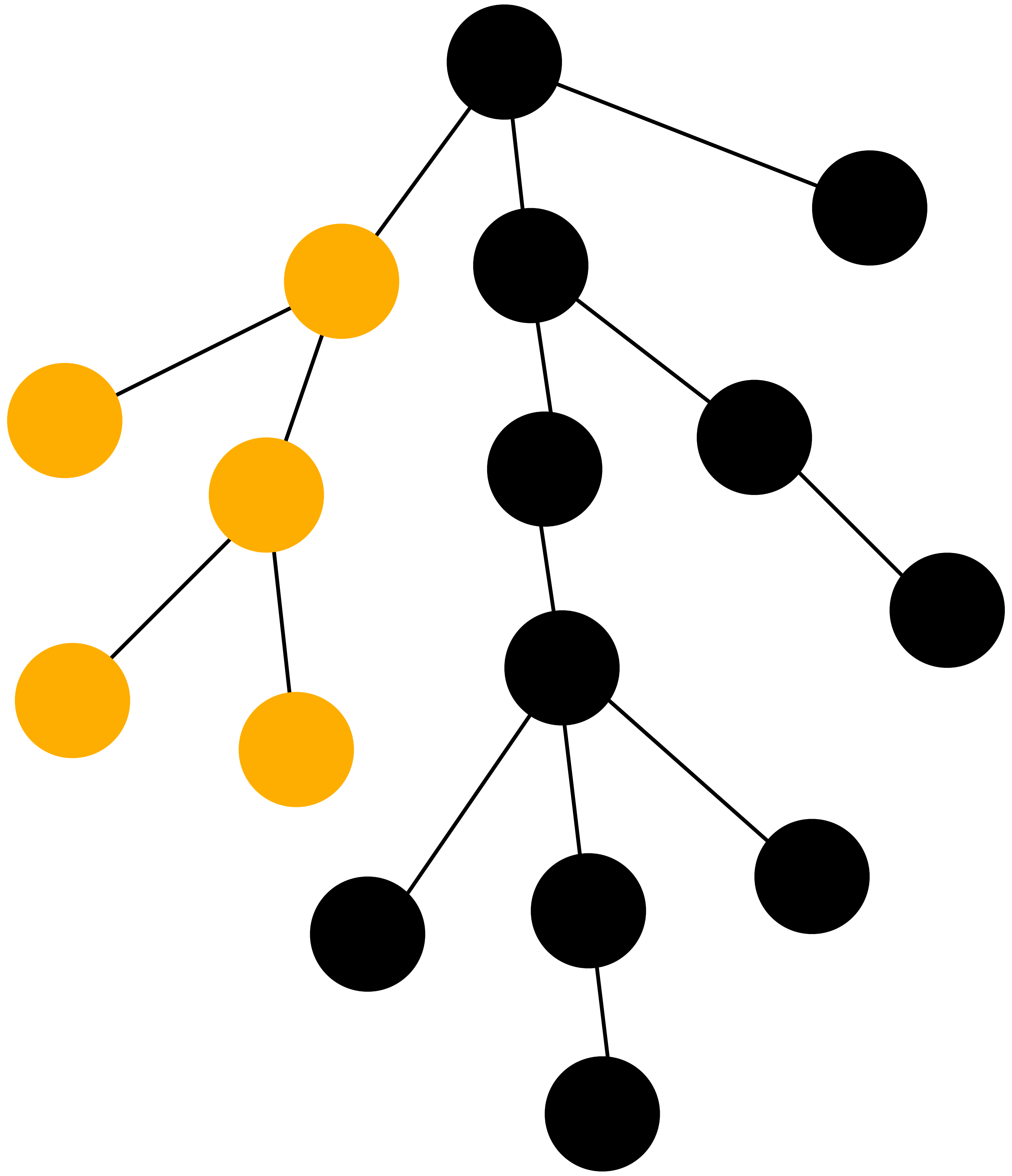
1. Change the value of node s to v
2. Find the sum of all nodes in the subtree of s



CSES 1137 - Subtree Queries

Given a tree consisting of N nodes, answer Q queries of the form:

1. Change the value of node s to v
2. Find the sum of all nodes in the subtree of s

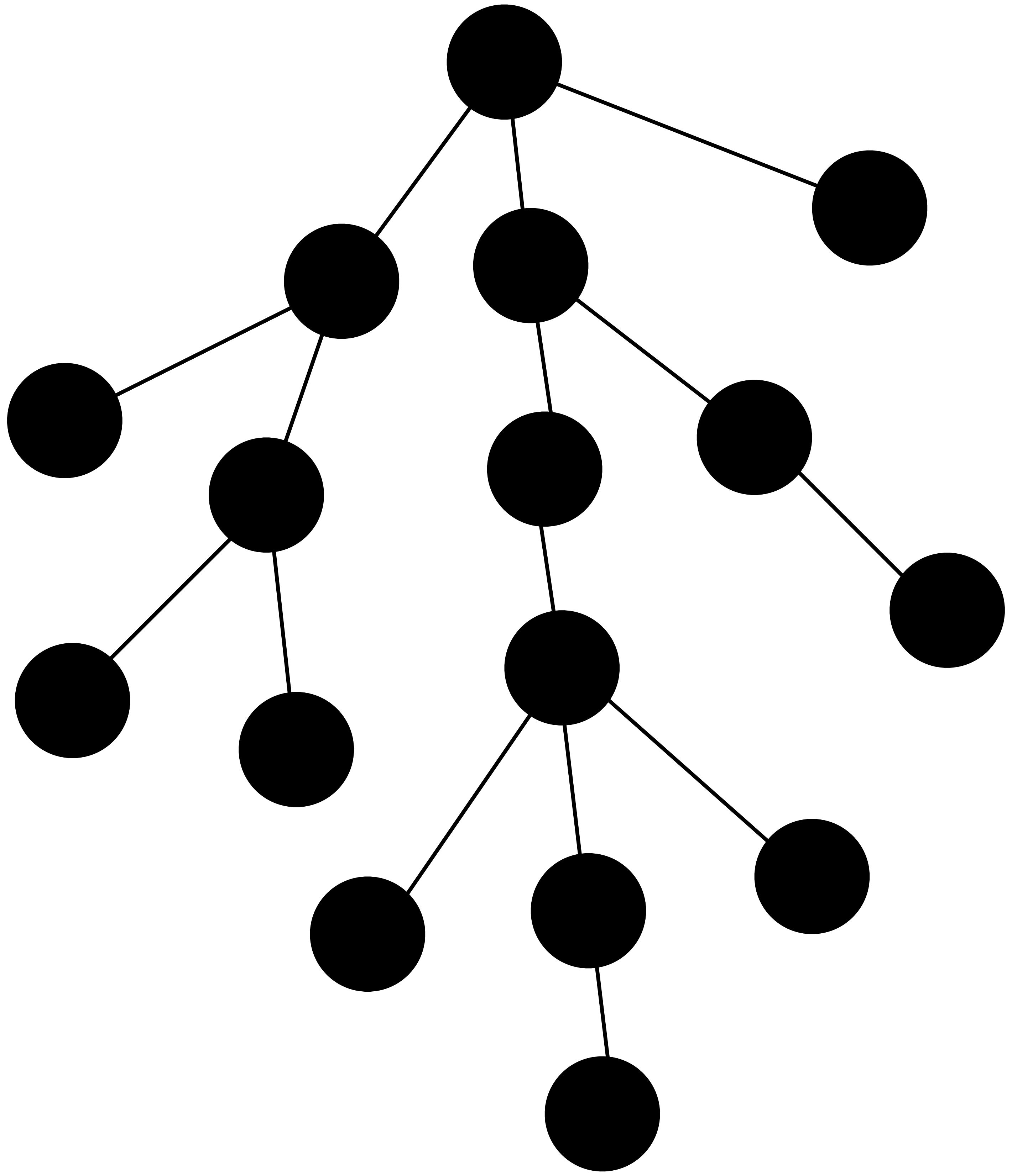


Solution: Just an Euler tour then a segment tree

CSES 2134 - Path Queries II

Given a tree consisting of N nodes, answer Q queries of the form:

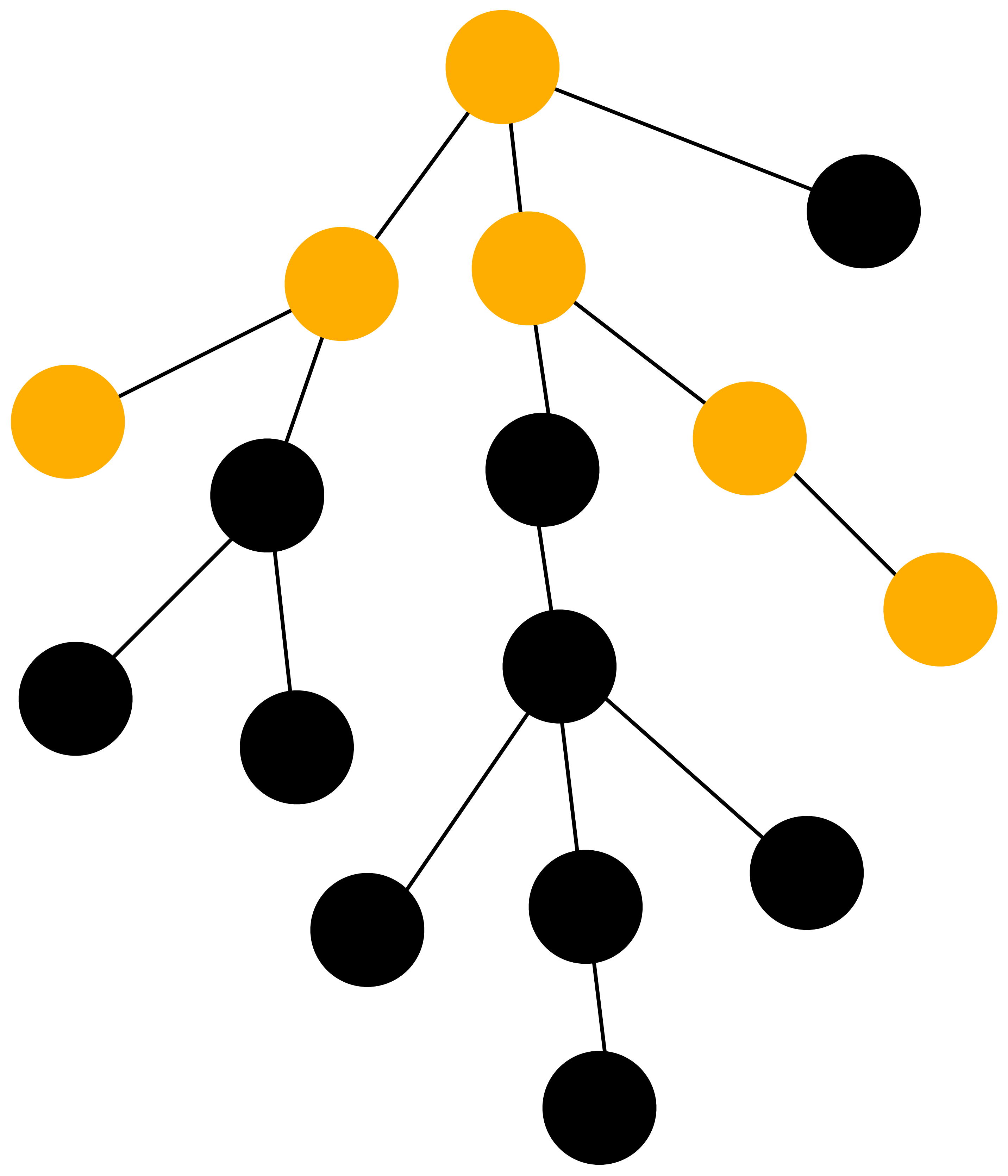
1. Change the value of node s to v
2. Find the maximum value of all nodes along the path between two nodes a, b



CSES 2134 - Path Queries II

Given a tree consisting of N nodes, answer Q queries of the form:

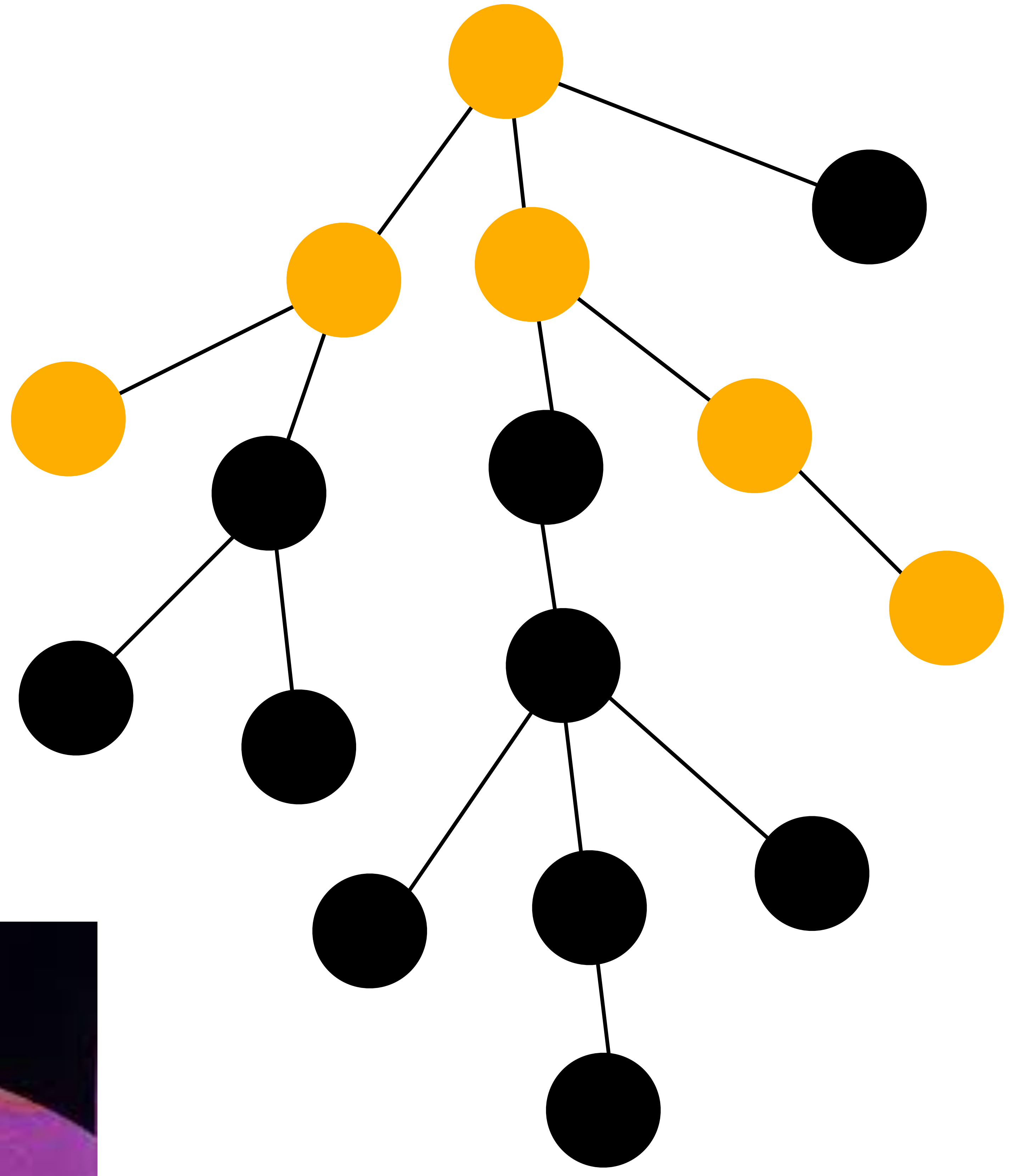
1. Change the value of node s to v
2. Find the maximum value of all nodes along the path between two nodes a, b



CSES 2134 - Path Queries II

Given a tree consisting of N nodes, answer Q queries of the form:

1. Change the value of node s to v
2. Find the maximum value of all nodes along the path between two nodes a, b



Solution: Not just an Euler tour then a segment tree

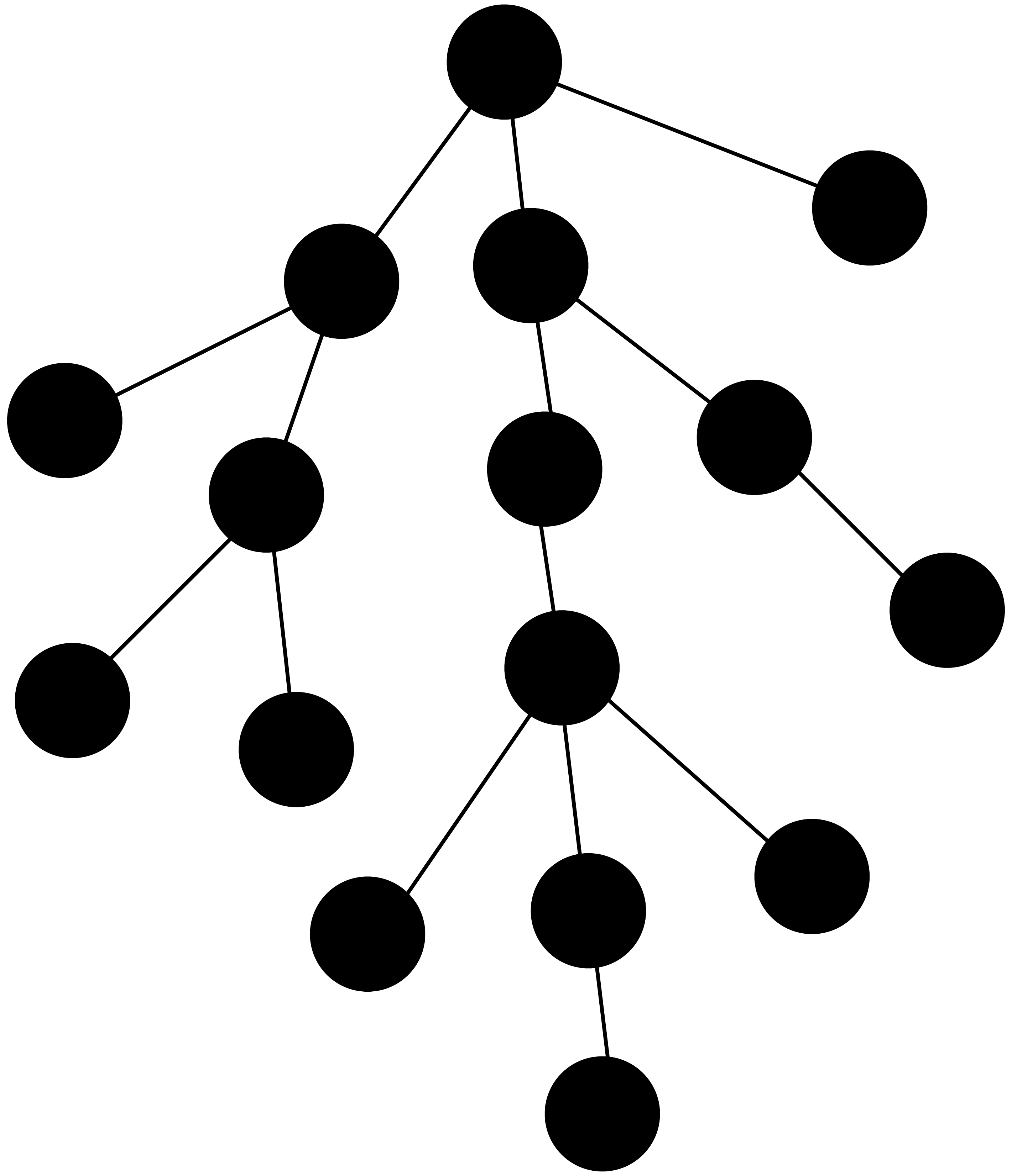
Basic Heavy-Light Decomposition

We start with some definitions:

Let $s(u)$ denote the size of the subtree of u .

Then v is a heavy child of $u \iff s(v) \geq \frac{s(u)}{2}$
All other children are called light.

We call the edge leading to a heavy child a heavy edge. All other edges are labeled light.



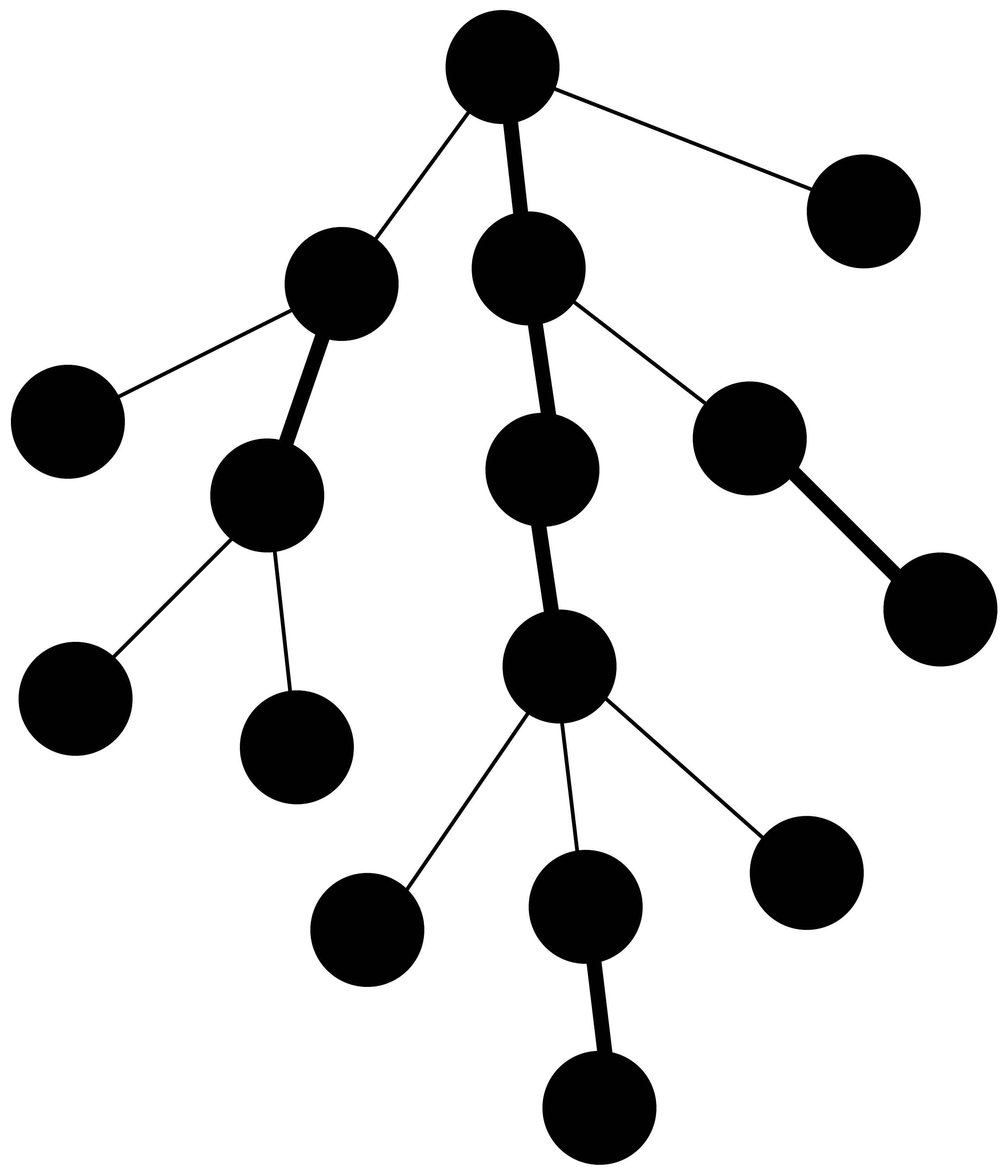
Basic Heavy-Light Decomposition

We start with some definitions:

Let $s(u)$ denote the size of the subtree of u .

Then v is a heavy child of $u \iff s(v) \geq \frac{s(u)}{2}$
All other children are called light.

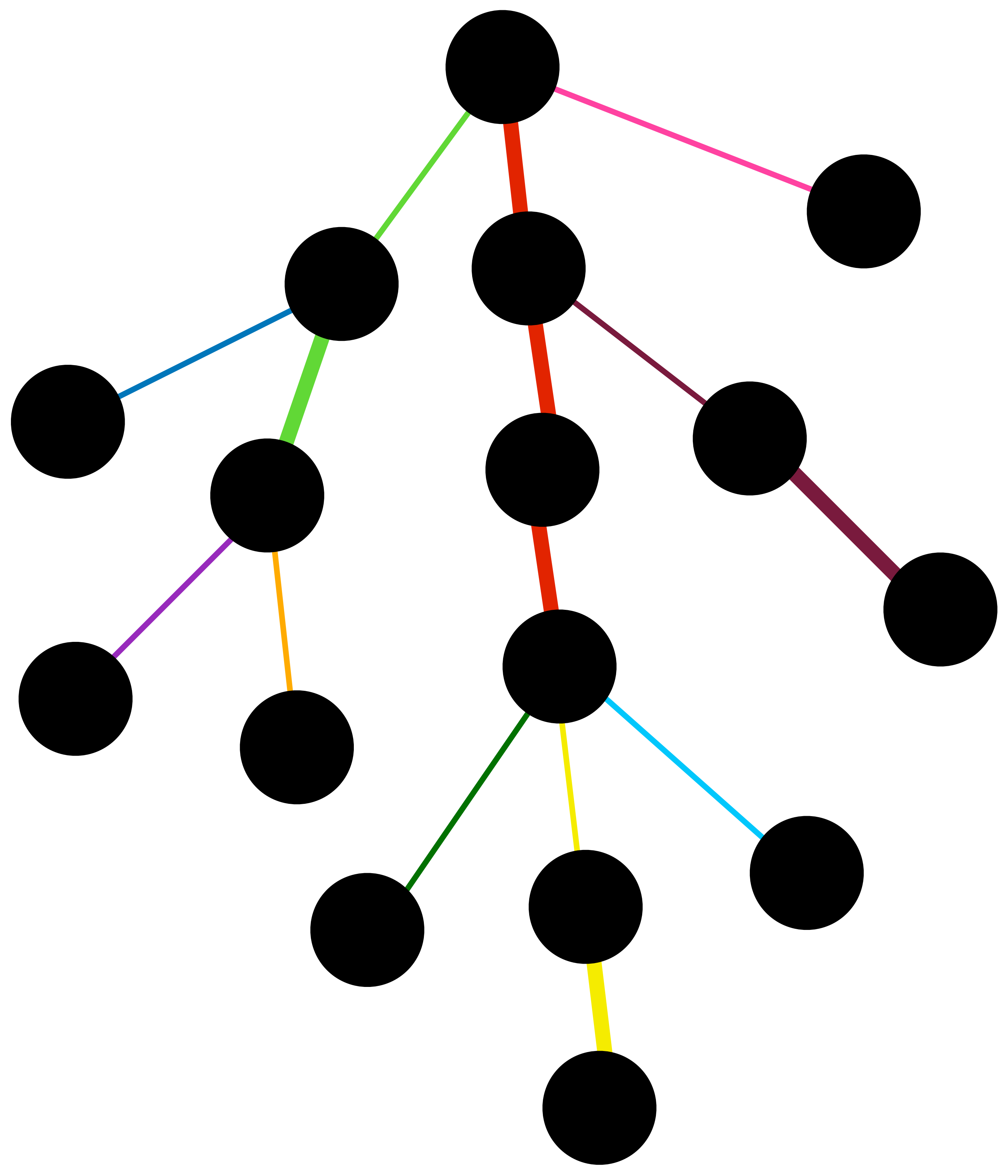
We call the edge leading to a heavy child a heavy edge. All other edges are labeled light.



An example of this decomposition: Heavy edges are thicker

Basic Heavy-Light Decomposition

We root the tree arbitrarily. Now consider all vertices which have no heavy children. We visit its parents until we encounter either the root node or a light edge. This splits the tree up into several paths on which we can process queries. We call these paths heavy paths. In the example the heavy paths are colored.

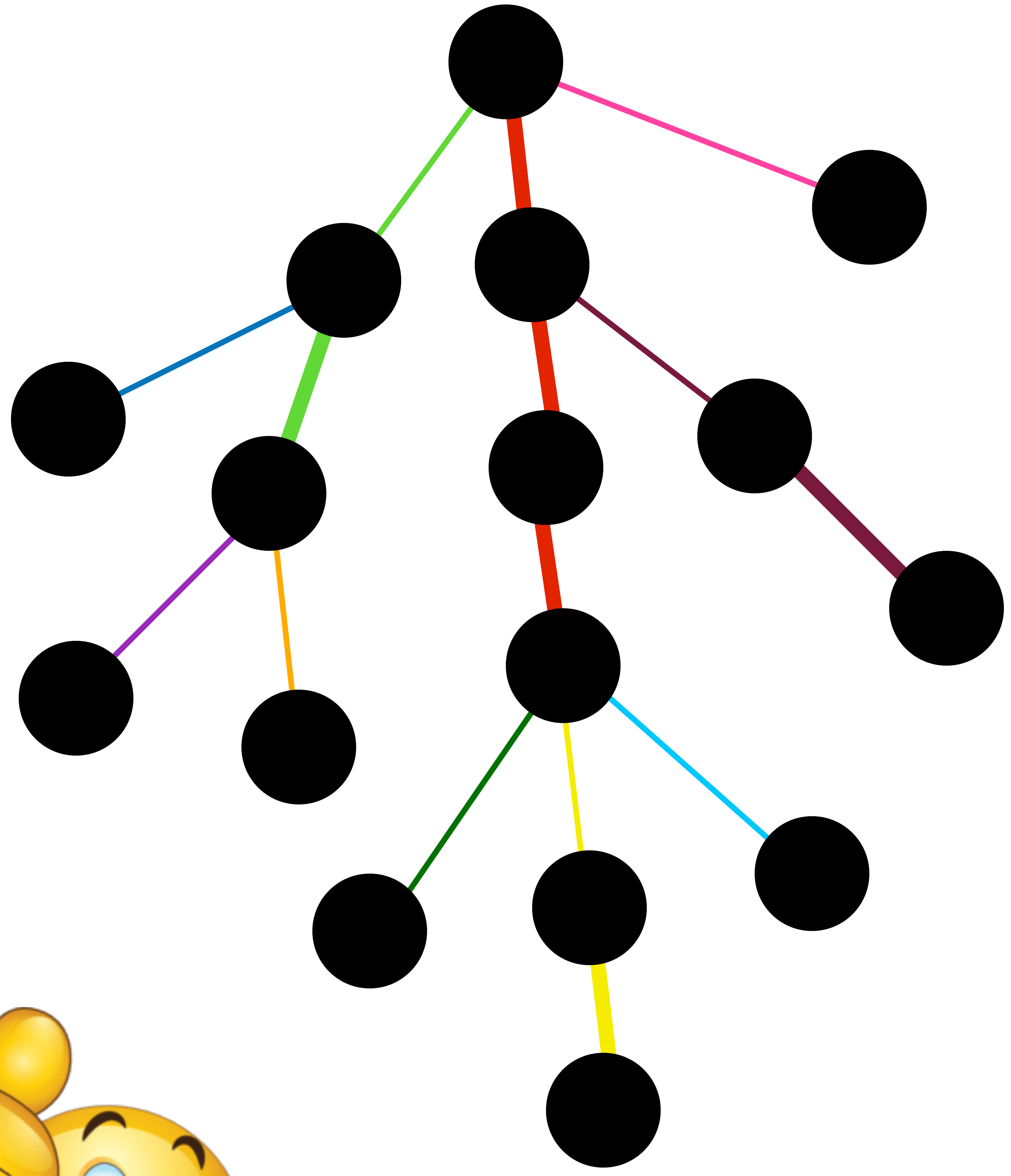


Importantly, this decomposition of our tree into paths satisfies two nice properties: All paths are disjoint, and the path from the root to any child uses no more than $O(\log n)$ paths.

We can now answer queries in $O(\log^2 n)$ time by building a segment tree on each path and using LCA queries!

Basic Heavy-Light Decomposition

We root the tree arbitrarily. Now consider all vertices which have no heavy children. We visit its parents until we encounter either the root node or a light edge. This splits the tree up into several paths on which we can process queries. We call these paths heavy paths. In the example the heavy paths are colored.



Importantly, this decomposition of our tree into paths satisfies two nice properties: All paths are disjoint, and the path from the root to any child uses no more than $O(\log n)$ paths.

We can now answer queries in $O(\log^2 n)$ time by building a segment tree on each path and using LCA queries!

Implementation

To make implementation easier, rather than using multiple segment trees, we combine them all into a single segment tree. We also do not separately answer LCA queries: We walk up to the LCA in a method similar to binary lifting. Finally, we change the definition of a heavy child to mean the child with the largest subtree, with ties broken arbitrarily. The basic structure of a HLD data structure:

```
struct hld {
  int n;
  vector<int> par, heavy, depth, root, pos;
  segtree st;

  int dfs(int u, int p, vector<vector<int>>& e);

  hld(int n, vector<int>& v, vector<vector<int>>& e);

  int query(int a, int b);

  void update(int n, int v);
};
```

Implementation

Initialization: DFS of the tree

```
int dfs(int u, int p, vector<vector<int>>& e) {
  int sz=1, m=0;
  for (auto v : e[u]) {
    if (v==p) continue;
    depth[v]=depth[u]+1, par[v]=u;
    int t=dfs(v, u, e);
    if (t>m) heavy[u]=v, m=t;
    sz+=t;
  }
  return sz;
}
```

Implementation

Initialization: Constructing the Heavy Paths

We loop through all nodes. If some node is not the heavy child of some other node, then the edge to its parent must be light, and therefore the node must be the root of a heavy path. We then keep descending to the heavy children until we reach a leaf. This ensures that all heavy paths are continuous sections of the array.

```
hld(int n, vector<int>& v, vector<vector<int>>& e) {
  this->n=n;
  heavy.assign(n+1, 0); depth.assign(n+1, 0);
  root.assign(n+1, 0); pos.assign(n+1, 0);
  par.assign(n+1, 0); par[1]=1;
  dfs(1, 1, e);
  vector<int> a;
  for (int i=1, t=1; i <= n; ++i) {
    if (i==1 || heavy[par[i]]!=i) {
      for (int j = i; j; j=heavy[j]) {
        root[j]=i;
        pos[j]=t++;
        a.push_back(v[j]);
      }
    }
  }
  st.init(n, a);
}
```

Implementation

Answering Queries

Path Queries

```
int query(int a, int b) {
  int s=0;
  while (root[a]!=root[b]) {
    if (depth[root[a]]>depth[root[b]]) swap(a, b);
    s=max(s, st.query(pos[root[b]], pos[b]));
    b=par[root[b]];
  }
  if (depth[a]>depth[b]) swap(a, b);
  s=max(s, st.query(pos[a], pos[b]));
  return s;
}
```

Update Queries

```
void update(int n, int v) {
  st.update(pos[n], v);
}
```

Implementation

Basic segment tree implementation (not important)

```
struct segtree {
   int n;
   vector<int> st;

   void init(int a, vector<int>& v) {
      n=a;
      st.assign(2*n, 0);
      for (int i = n; i < 2*n; ++i) st[i]=v[i-n];
      for (int i = n-1; i > 0; --i) st[i]=max(st[2*i],st[2*i+1]);
   }

   int query(int a, int b) {
      int s=0;
      a+=n-1; b+=n-1;
      while (a<=b) {
         if (a%2==1) s=max(s, st[a++]);
         if (b%2==0) s=max(s, st[b--]);
         a/=2; b/=2;
      } return s;
   }

   void update(int p, int v) {
      for (st[p+=n-1]=v; p>1; p/=2) st[p/2]=max(st[p],st[p^1]);
   }
};
```

CSES 1735 - Range Updates and Sums

Given an array of N elements, answer Q queries of the form:

1. Increase each value in the range $[a, b]$ by v
2. Set each value in the range $[a, b]$ to v
3. Find the sum of all values in the range $[a, b]$

CSES 1735 - Range Updates and Sums

Given an array of N elements, answer Q queries of the form:

1. Increase each value in the range $[a, b]$ by v
2. Set each value in the range $[a, b]$ to v
3. Find the sum of all values in the range $[a, b]$

Solution: Lazy Segment trees

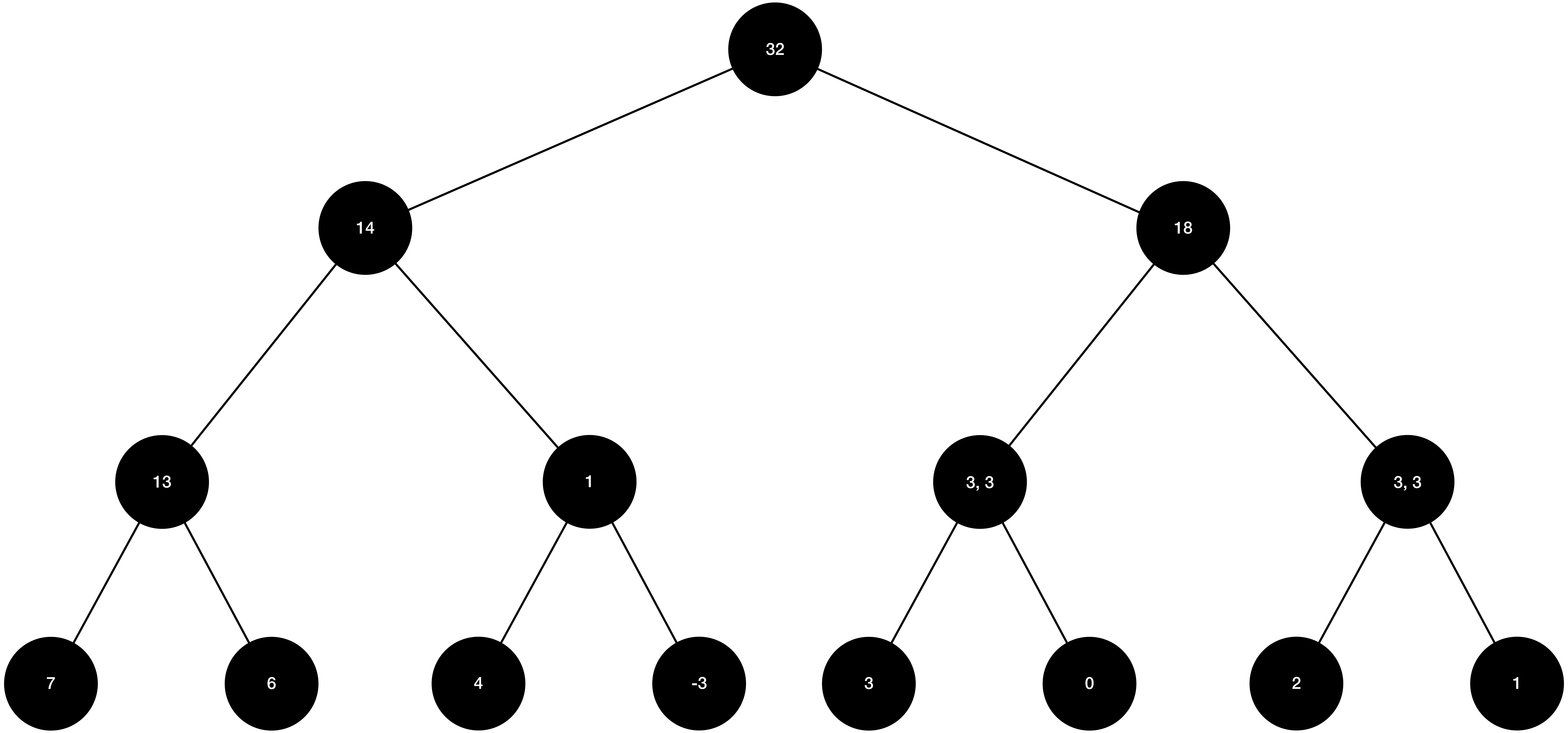
Lazy Segment Trees

When updating a range, we don't immediately update the segment tree: We only update when strictly necessary. At each node s, we store the value $lz[s]$ that stores a pending update that has yet to be pushed down to its children. When it becomes necessary to update a node, we push down the update to its children where the update can remain pending until it needs to be pushed down further.

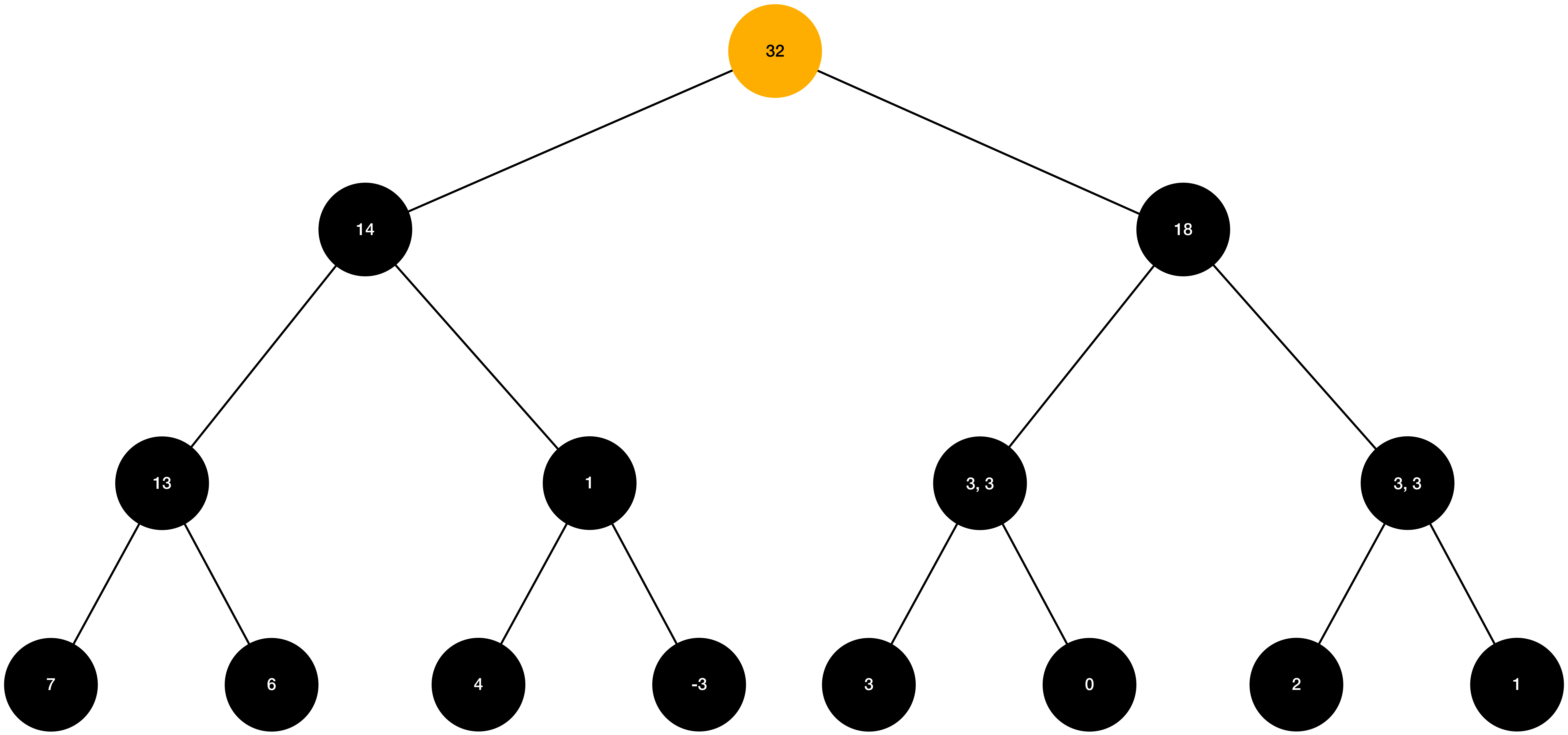
We are therefore required to implement a push function that takes a node and propagates an update to its children. However, a child could already have a pending update, and we cannot simply overwrite it. We therefore need a second function that takes two pending updates and combines it into one.

Example of Lazy Propagation

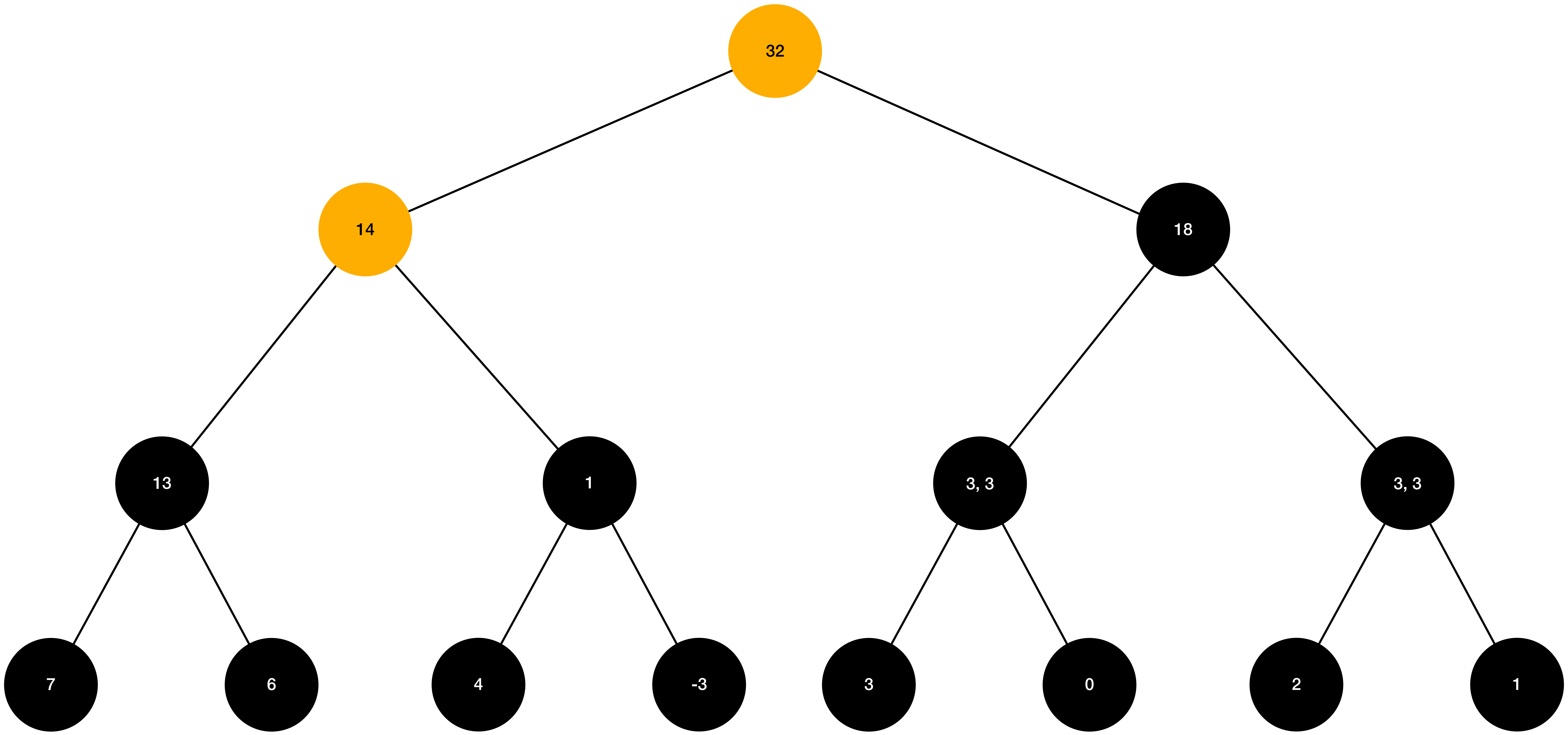
Add 7 to the range [3, 8]



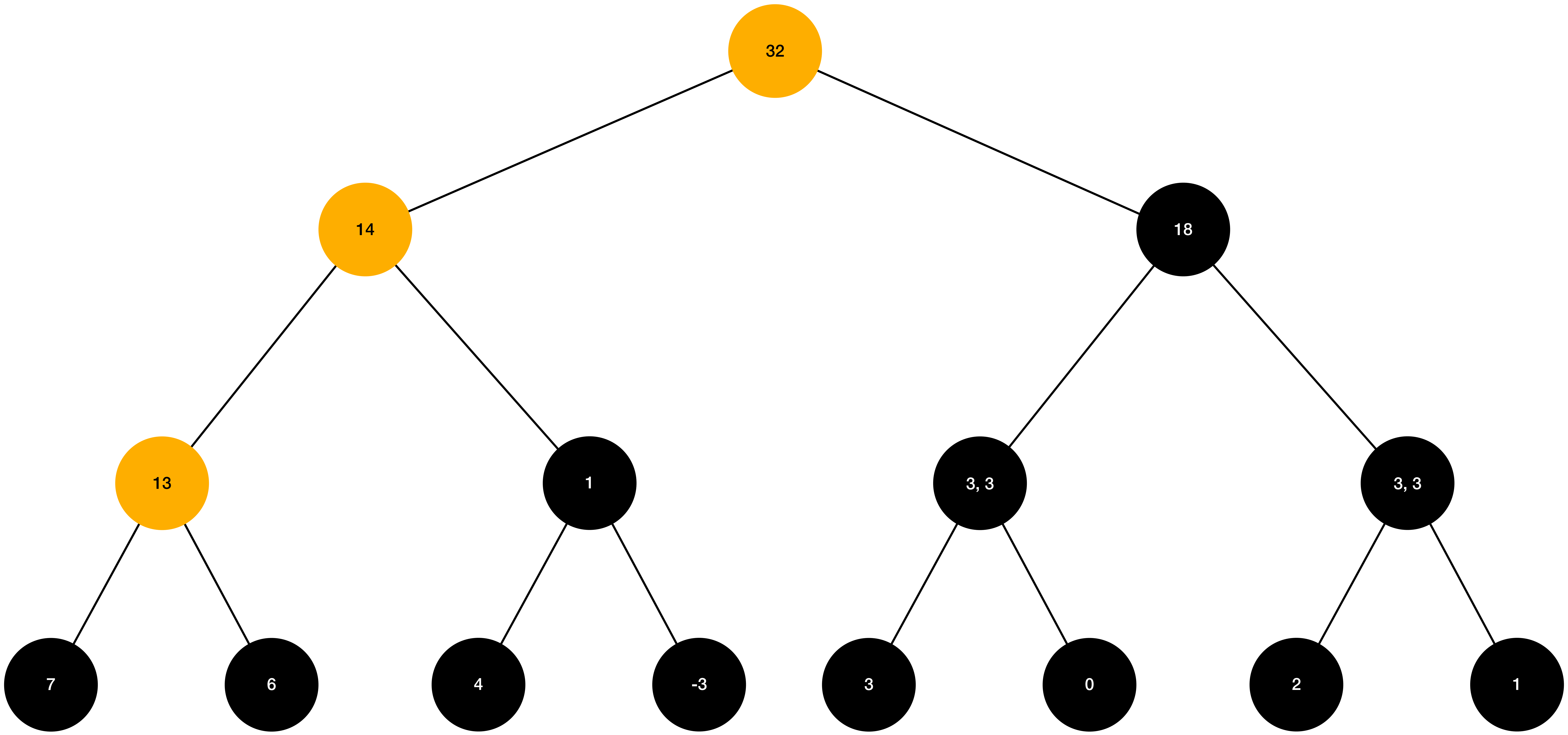
Example of Lazy Propagation



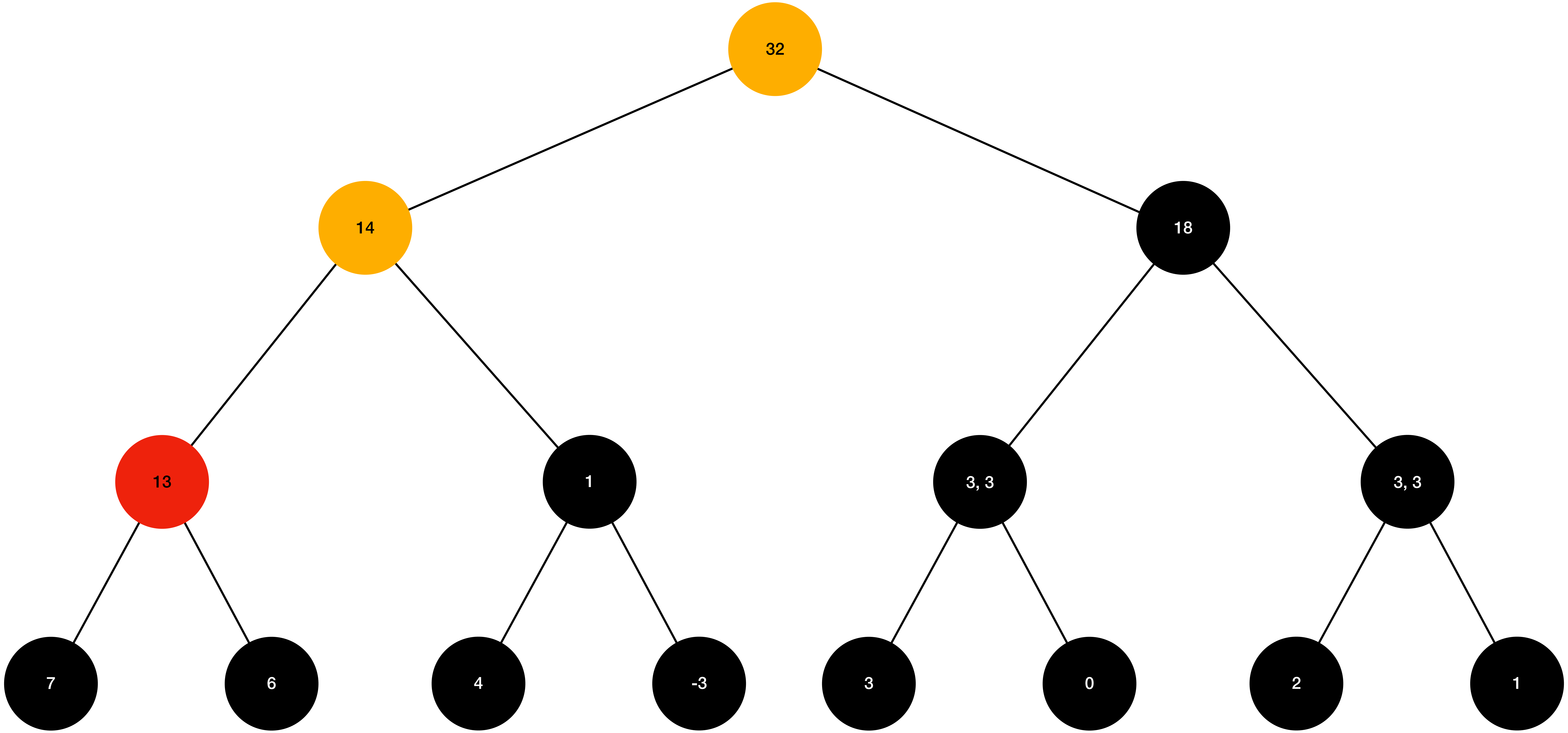
Example of Lazy Propagation



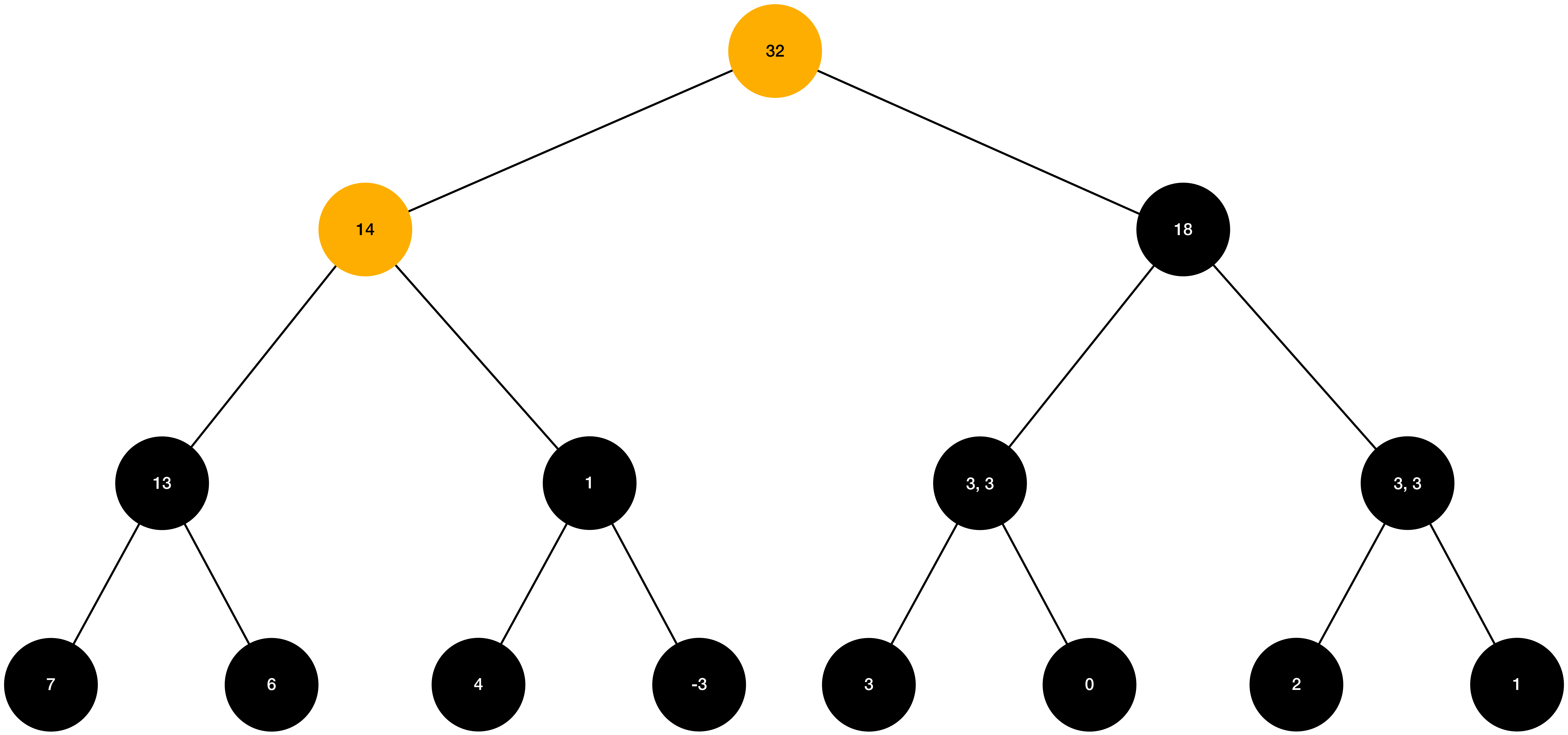
Example of Lazy Propagation



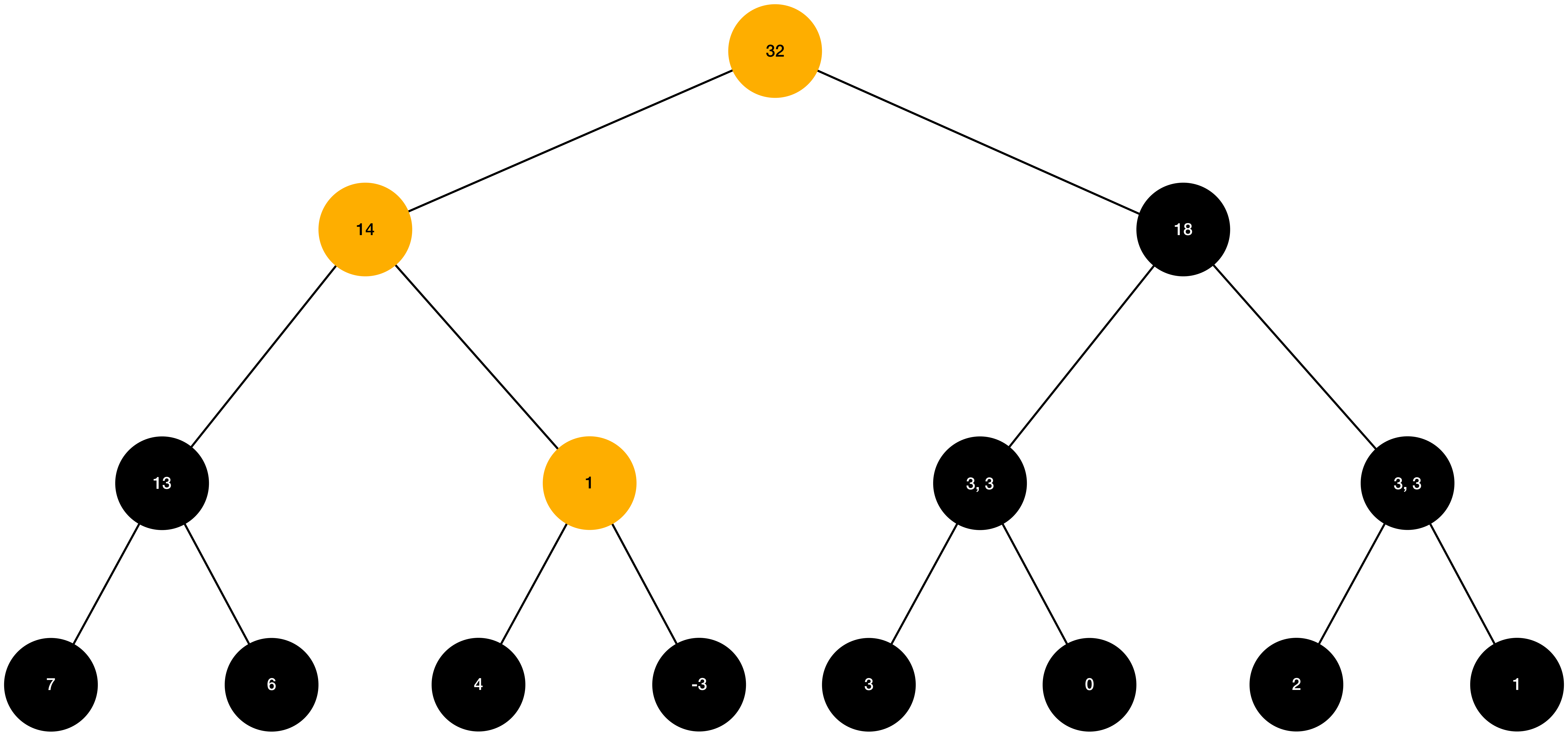
Example of Lazy Propagation



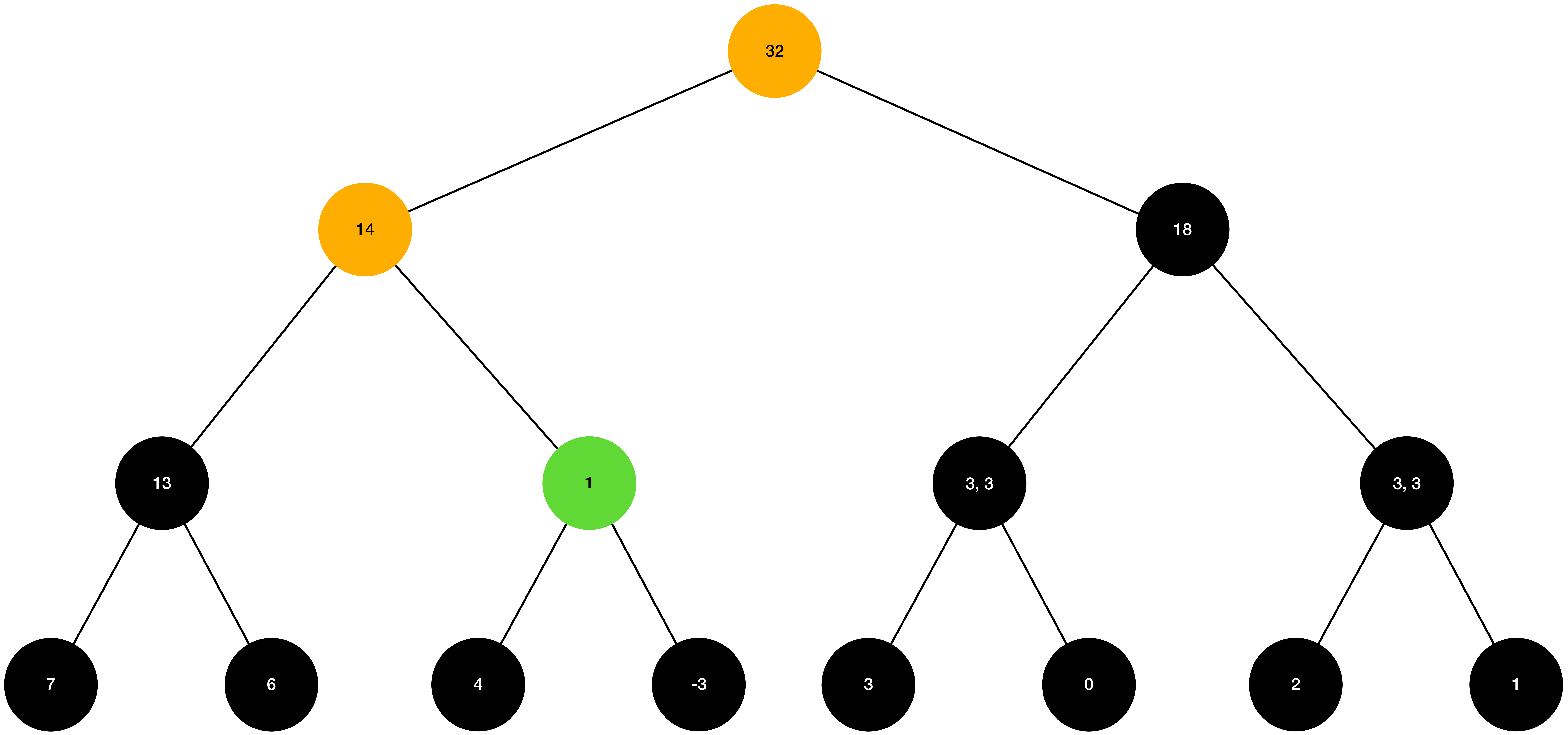
Example of Lazy Propagation



Example of Lazy Propagation

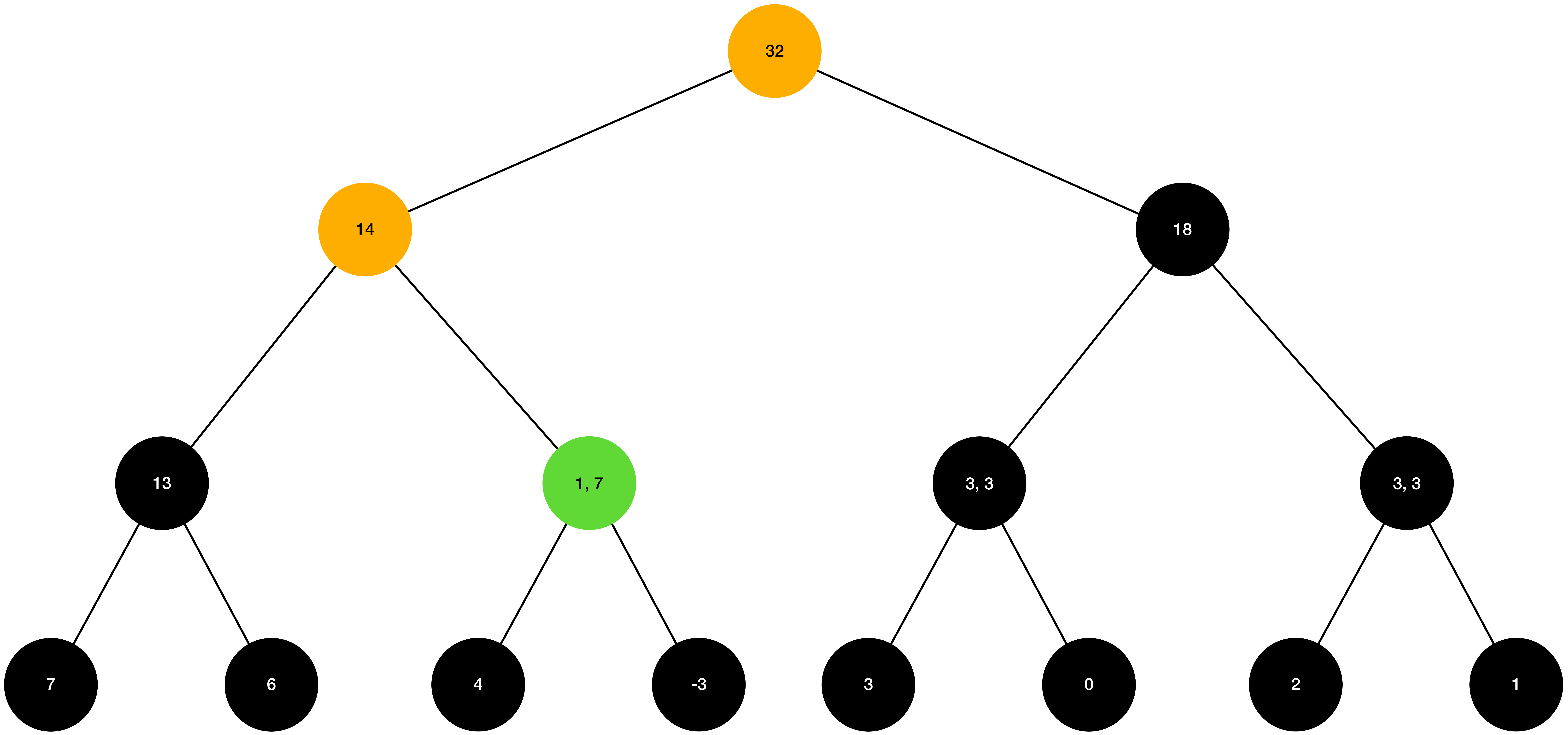


Example of Lazy Propagation



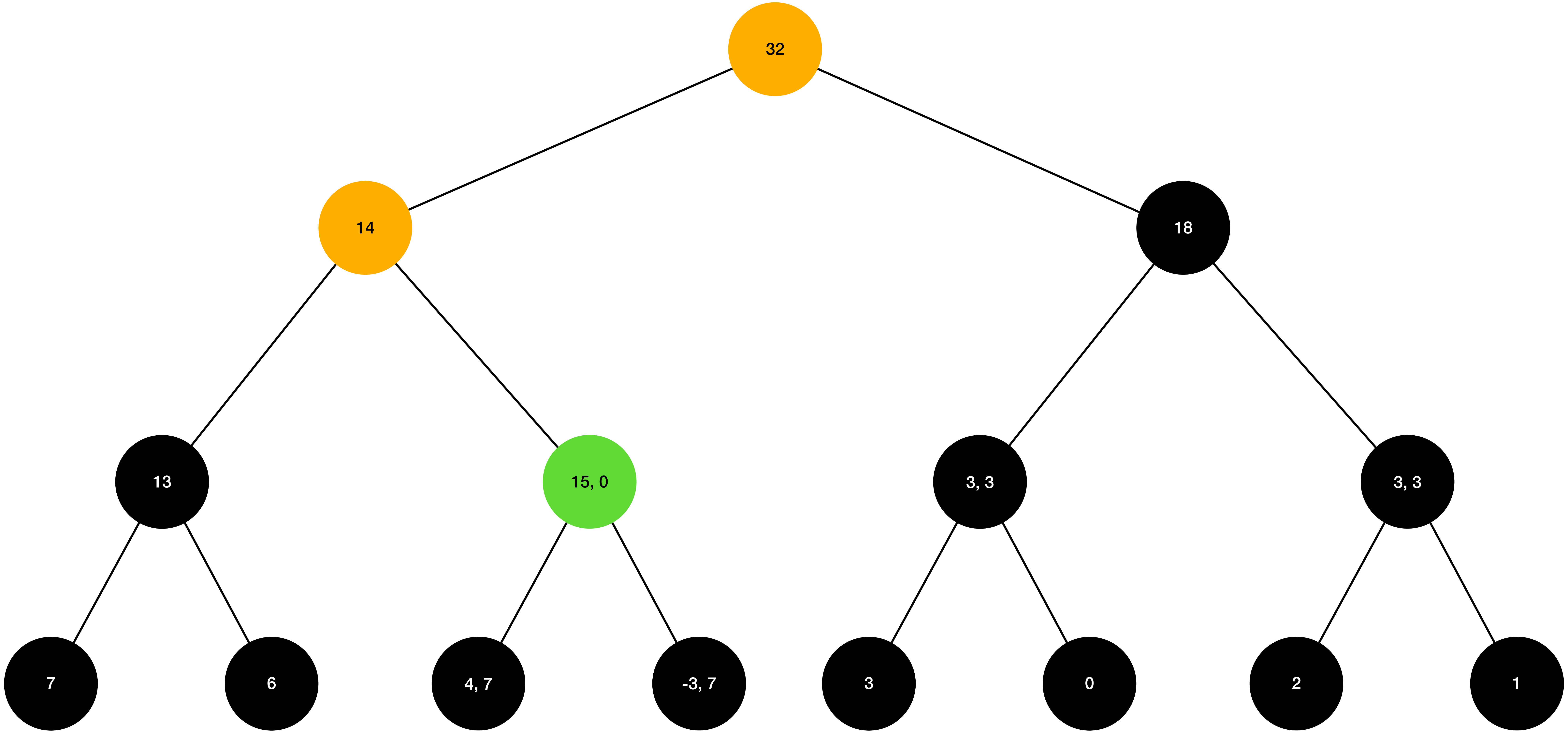
Example of Lazy Propagation

Add lazy update

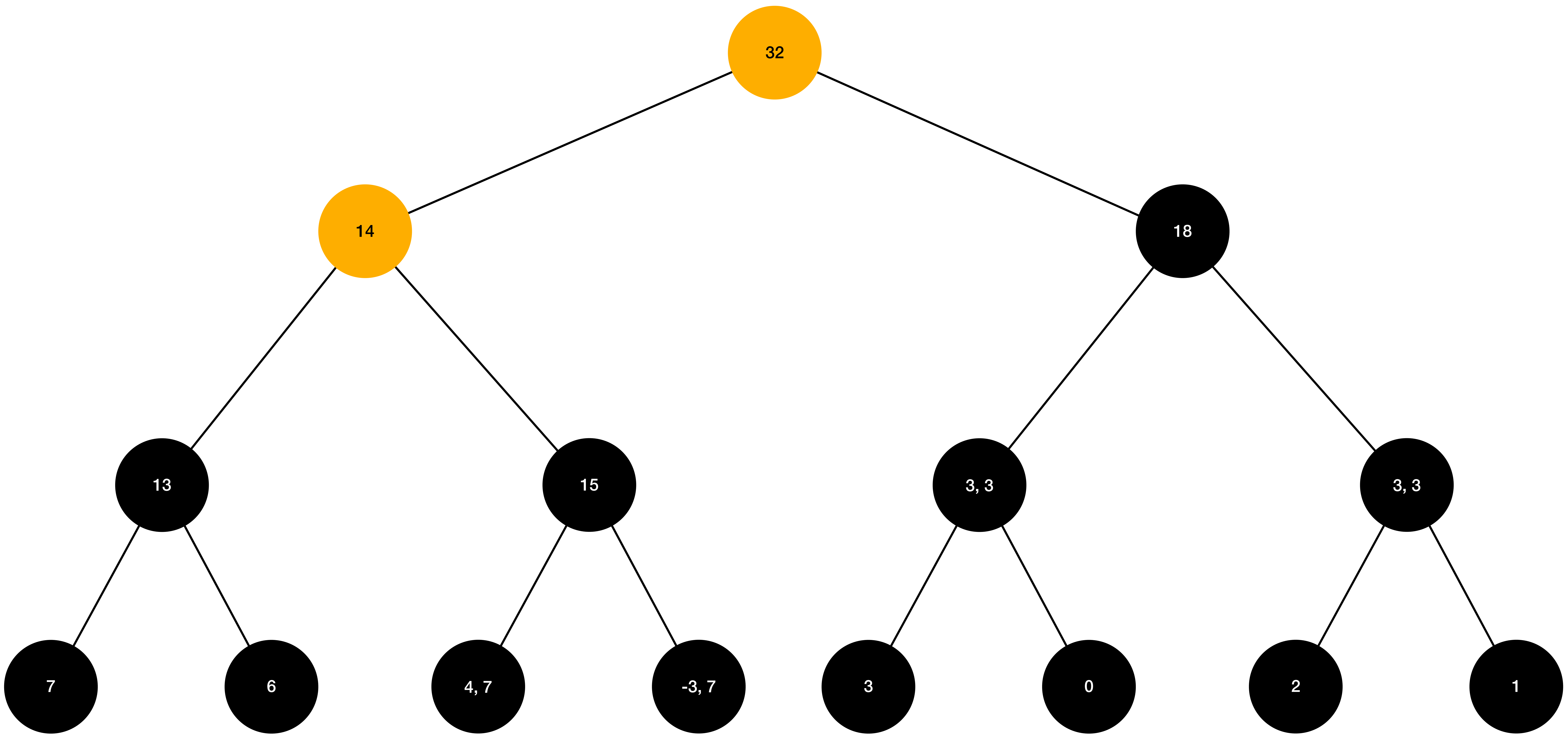


Example of Lazy Propagation

Push lazy update down

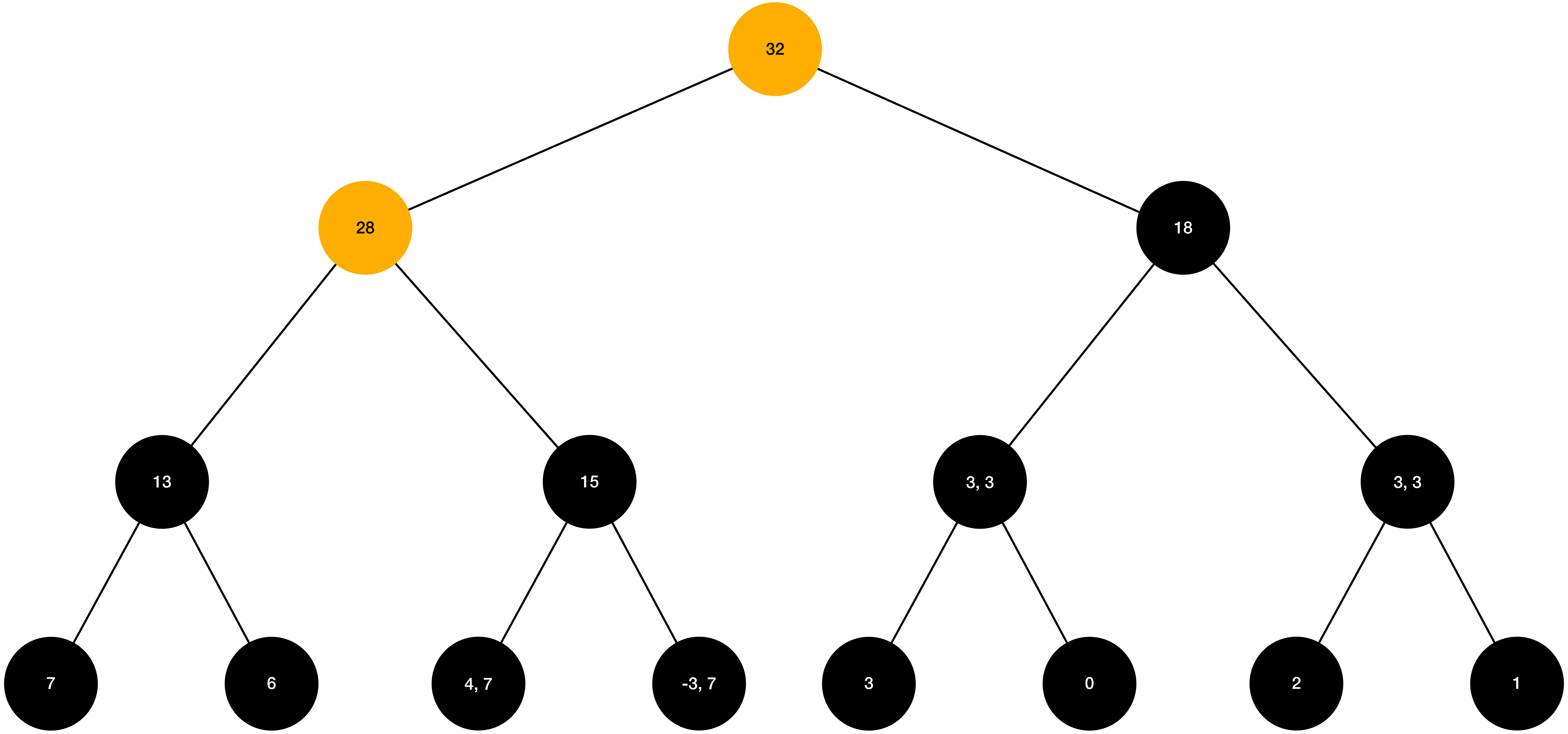


Example of Lazy Propagation

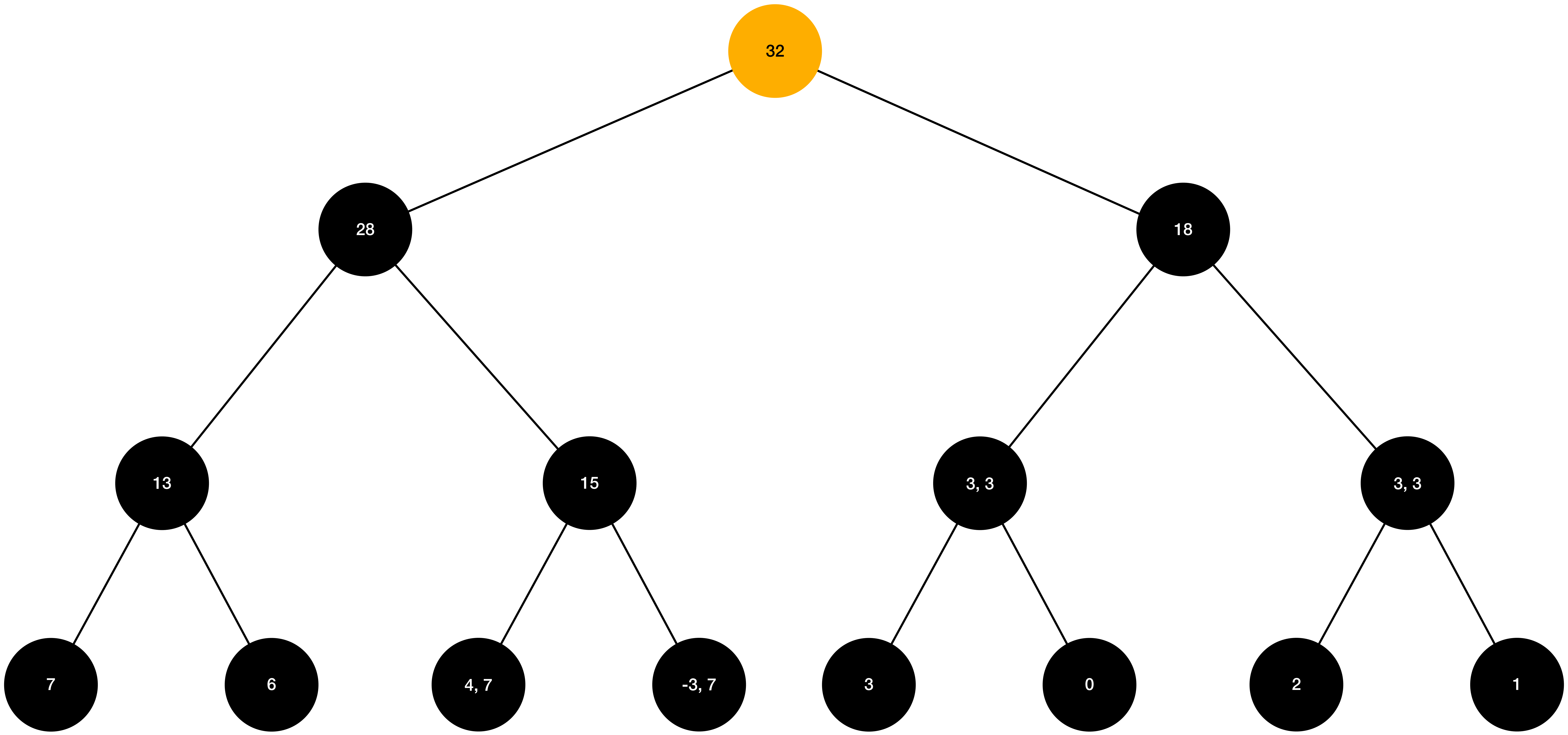


Example of Lazy Propagation

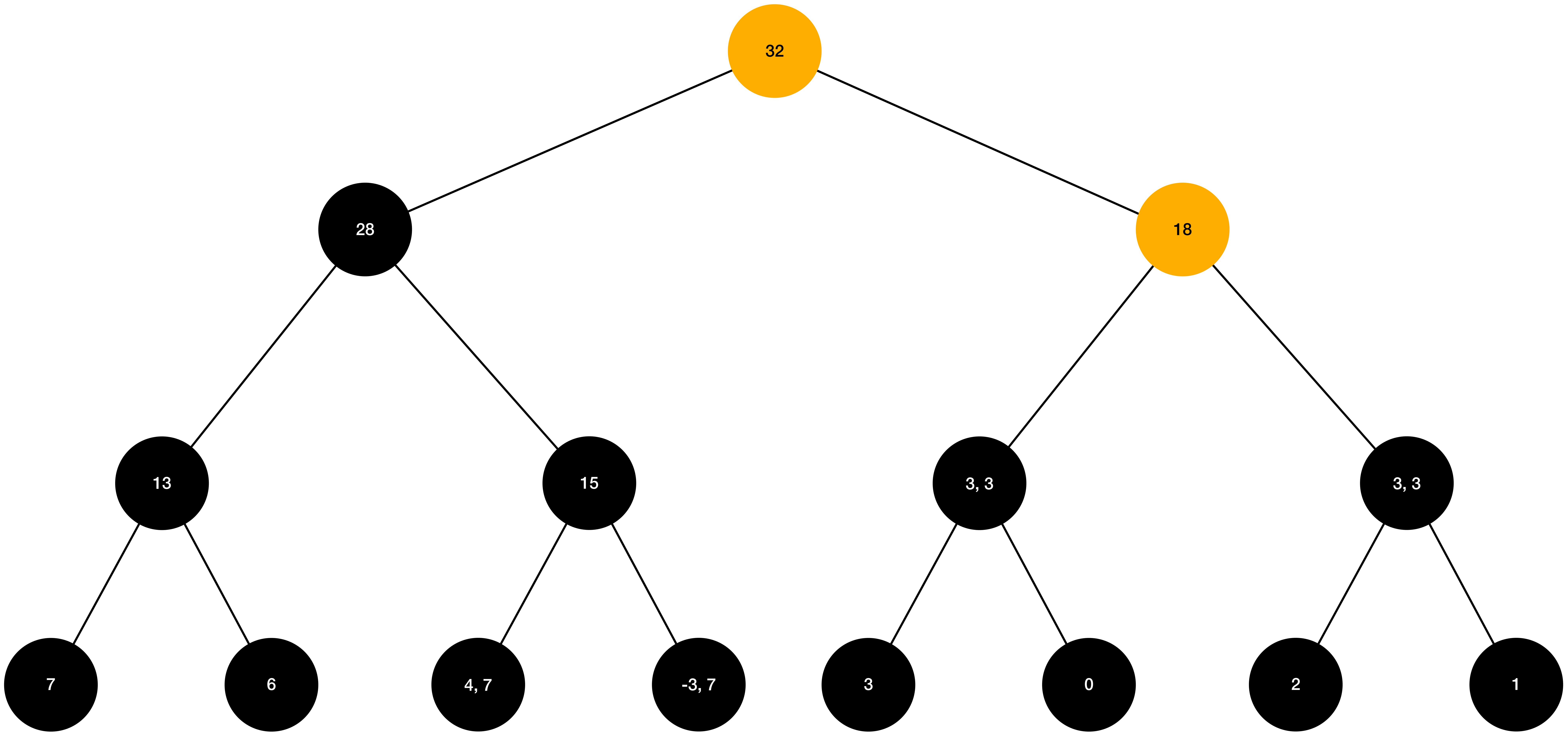
Recalculate node



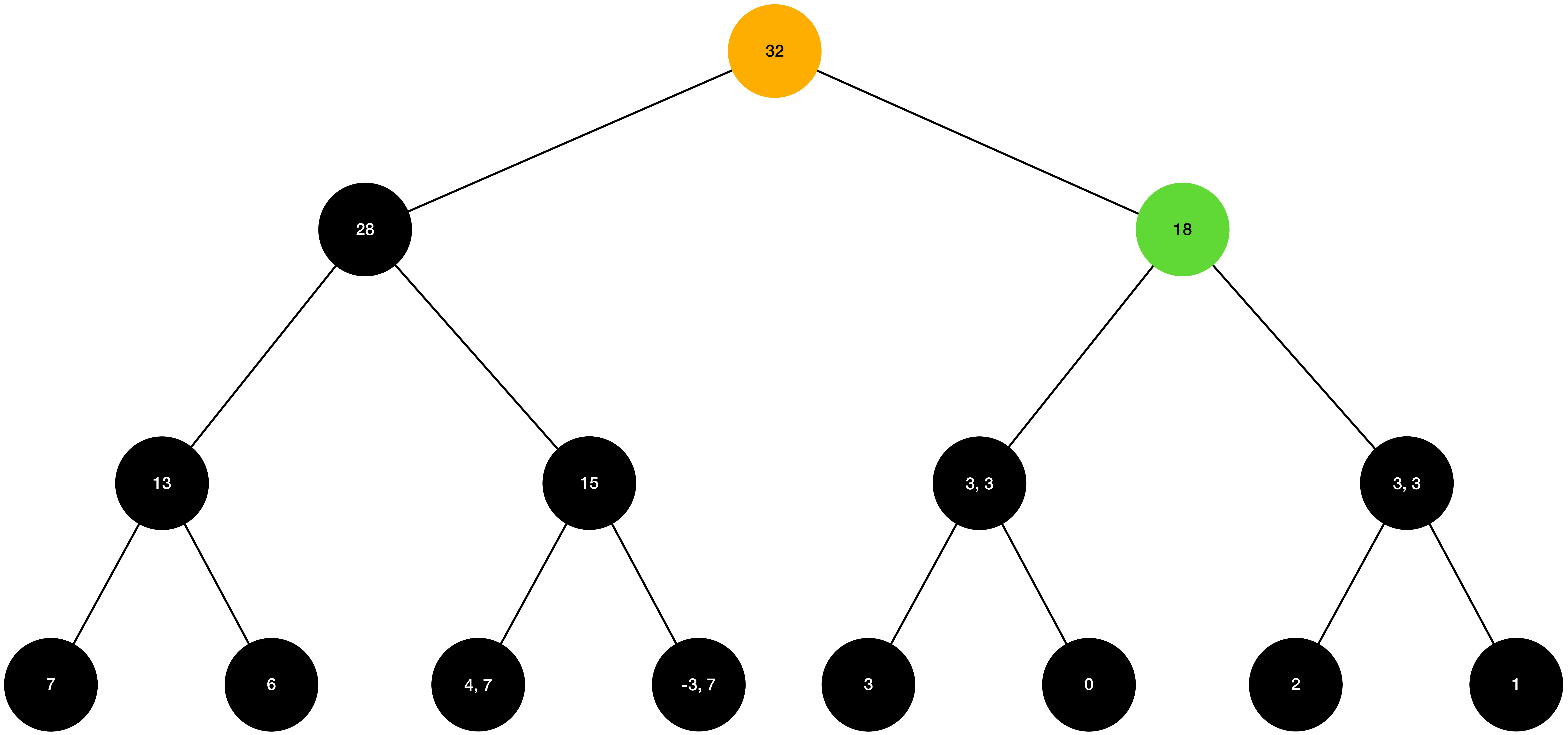
Example of Lazy Propagation



Example of Lazy Propagation

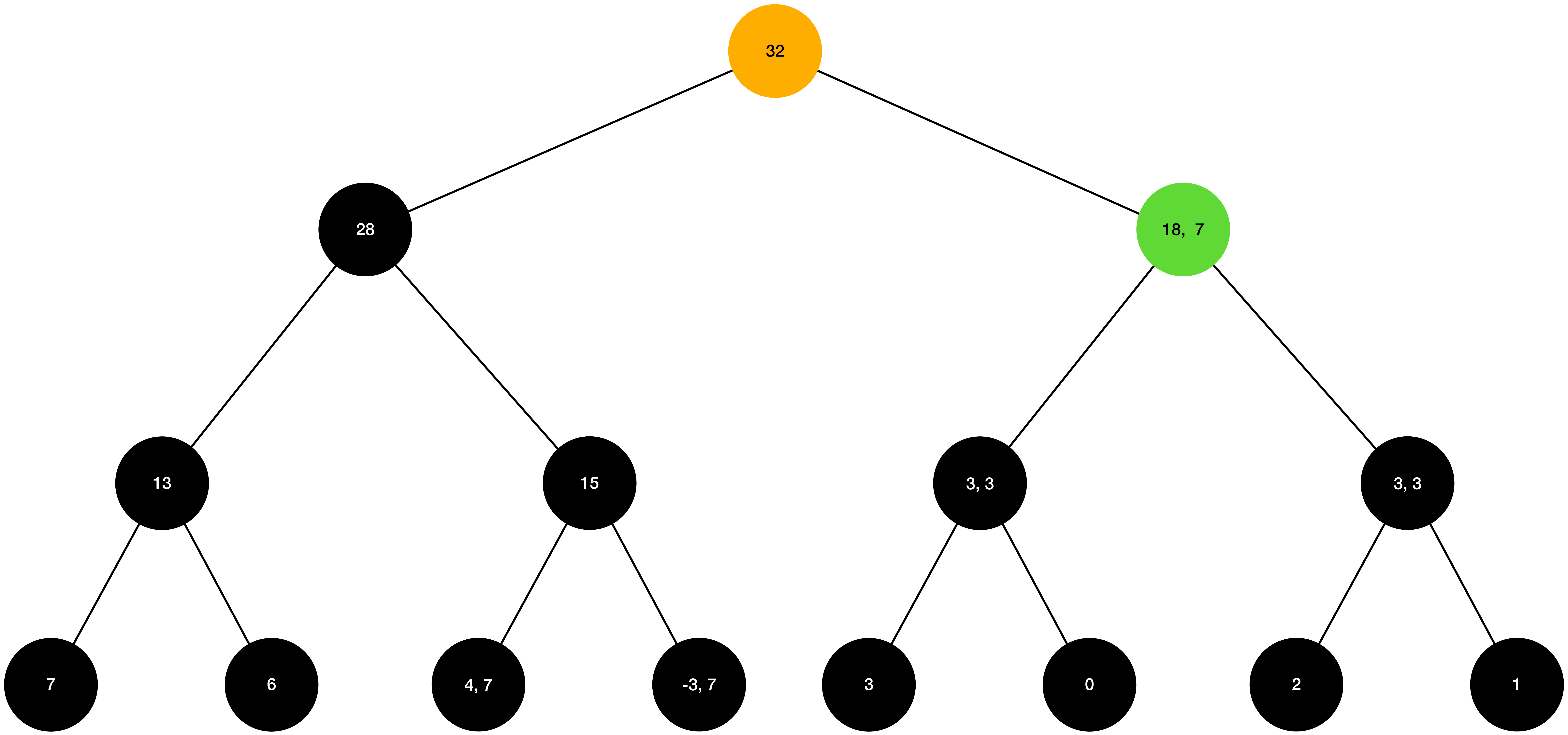


Example of Lazy Propagation



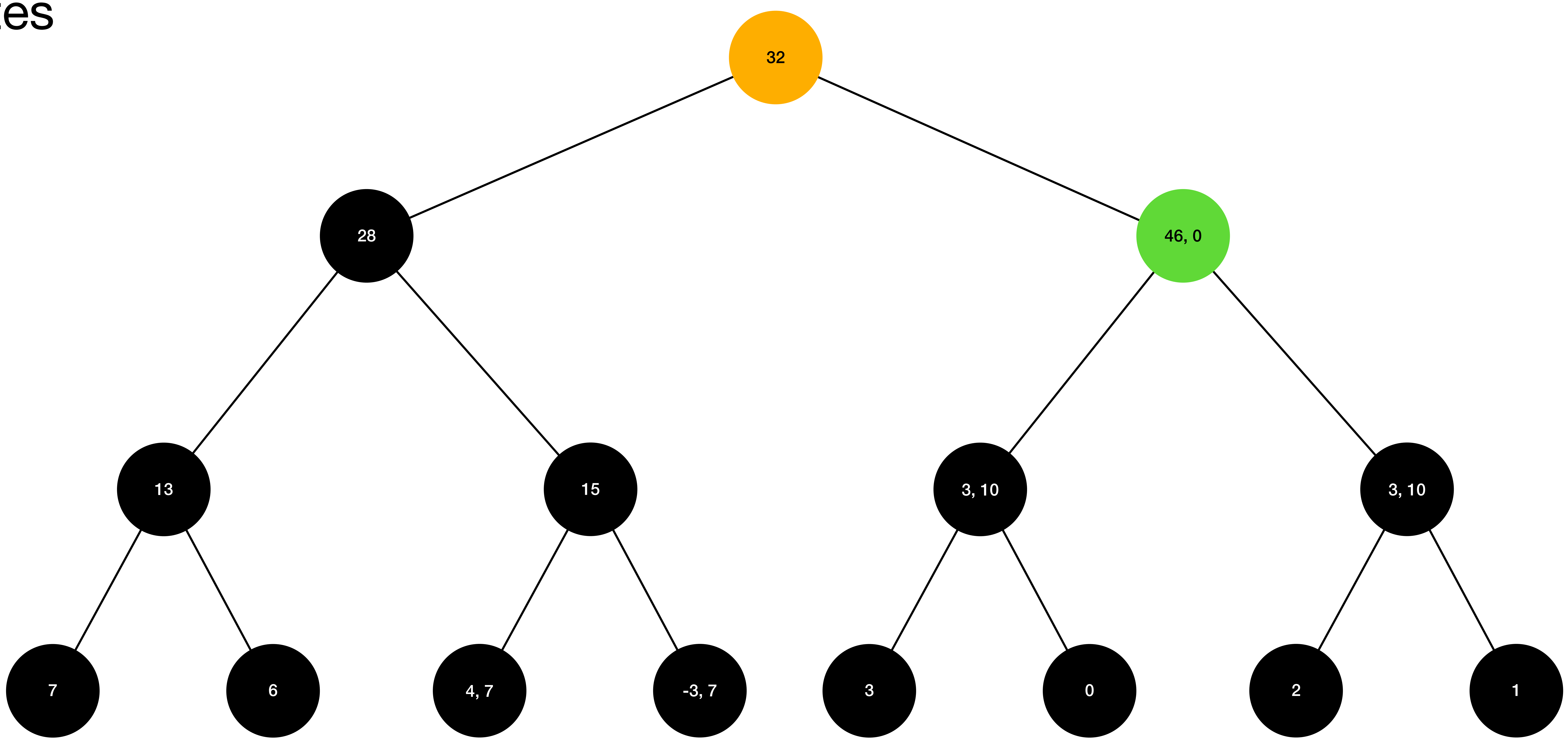
Example of Lazy Propagation

Add lazy update

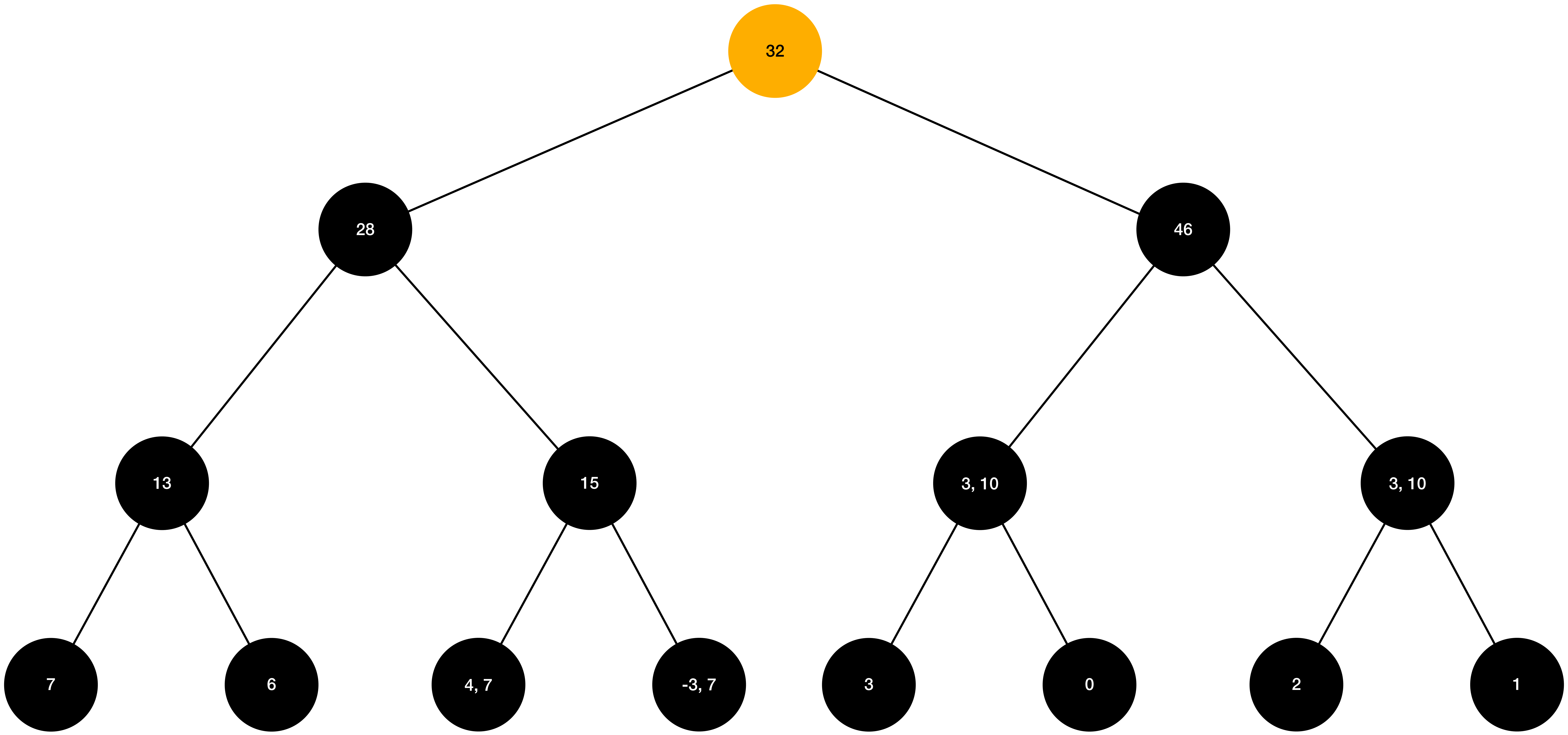


Example of Lazy Propagation

Push lazy update down, combine updates

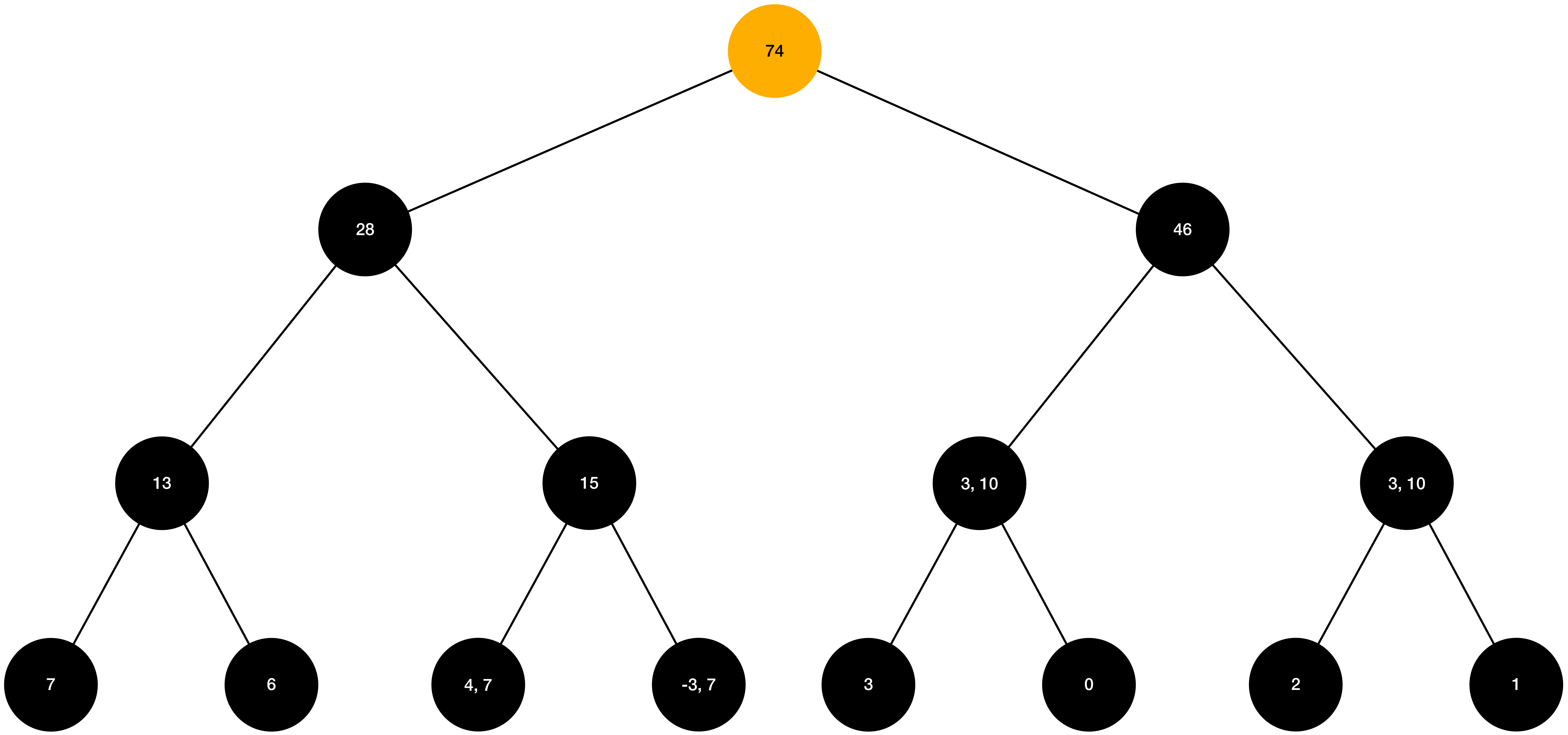


Example of Lazy Propagation

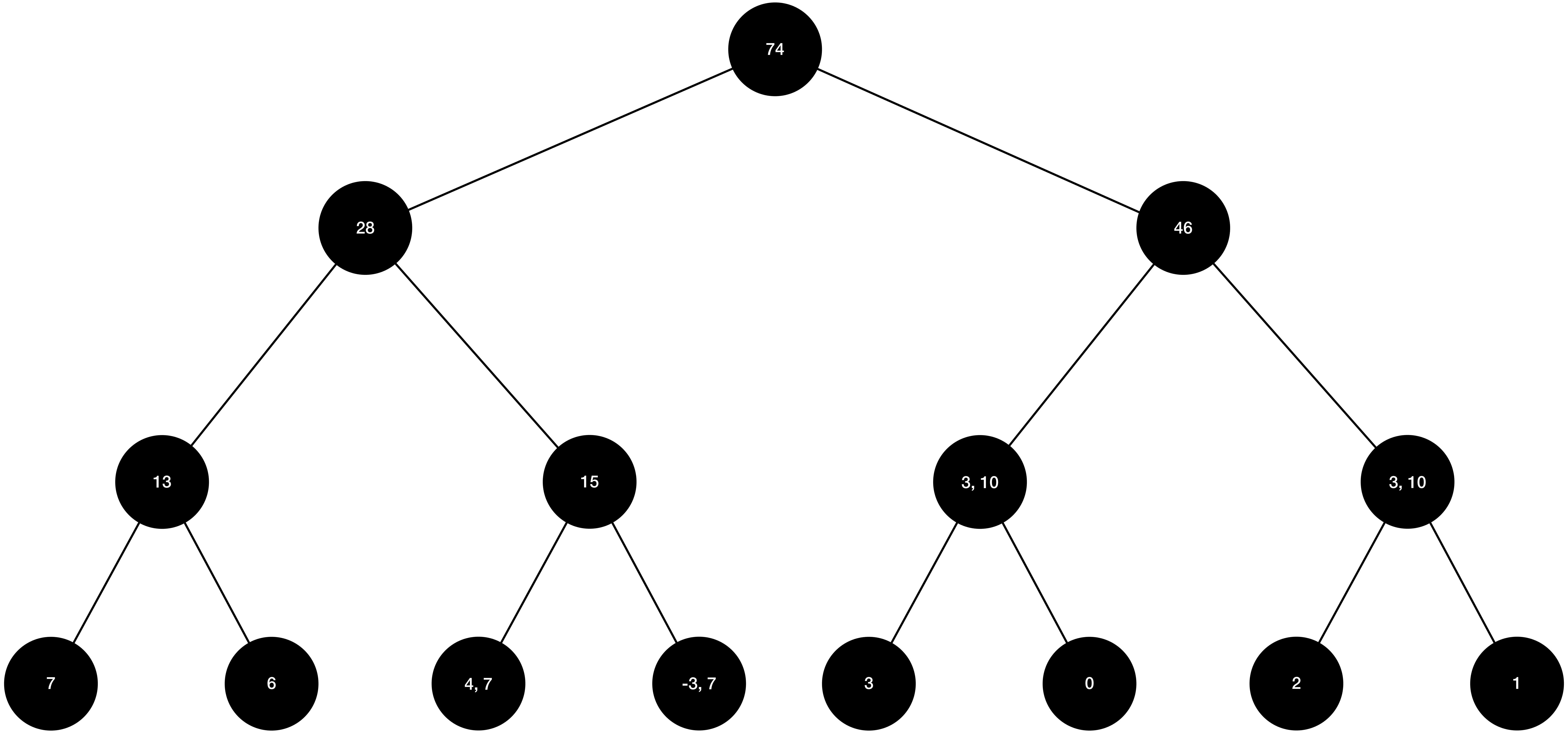


Example of Lazy Propagation

Recalculate node



Example of Lazy Propagation



Implementation

Not important here but worth mentioning

```
struct lazy {
   int n;
   vector<int> st, lza, lzs;

   lazy(int a) {
      n=a;
      while (n&(n-1)) n++;
      vector<int> b(2*n), c(2*n, 0), d(2*n, 0);
      for (int i = n; i < n+a; ++i) cin >> b[i];
      for (int i = n-1; i > 0; --i) b[i]=b[2*i]+b[2*i+1];
      swap(st, b); swap(lza, c); swap(lzs, d);
   }

   void push(int n, int l, int r) {
      if (lzs[n]) {
         st[n]=(r-l+1)*(lzs[n]+lza[n]);
         if (n<this->n) {
            lzs[2*n]=lzs[2*n+1]=lzs[n]+lza[n];
            lza[2*n]=lza[2*n+1]=0;
         } lzs[n]=lza[n]=0;
      } else {
         st[n]+=(r-l+1)*lza[n];
         if (n<this->n) {
            lza[2*n]+=lza[n];
            lza[2*n+1]+=lza[n];
         } lza[n]=0;
      }
   }

void add(int a, int b, int v, int n=1, int l=1, int r=-1) {
      if (n==1) r=this->n;
      push(n, l, r);
      if (r<a || b<l) return;
      if (a<=l && r<=b) { lza[n]+=v; push(n, l, r); return; }
      st[n]+=(min(b, r)-max(a, l)+1)*v;
      add(a, b, v, 2*n, l, (l+r)/2);
      add(a, b, v, 2*n+1, (l+r)/2+1, r);
   }

   void set(int a, int b, int v, int n=1, int l=1, int r=-1) {
      if (n==1) r=this->n;
      push(n, l, r);
      if (r<a || b<l) return;
      if (a<=l && r<=b) { lzs[n]=v; lza[n]=0; push(n, l, r); return; }
      set(a, b, v, 2*n, l, (l+r)/2);
      set(a, b, v, 2*n+1, (l+r)/2+1, r);
      st[n]=st[2*n]+st[2*n+1];
   }

   int query(int a, int b, int n=1, int l=1, int r=-1) {
      if (n==1) r=this->n;
      push(n, l, r);
      if (r<a || b<l) return 0;
      if (a<=l && r<=b) return st[n];
      return query(a, b, 2*n, l, (l+r)/2) + query(a, b, 2*n+1, (l+r)/2+1, r);
   }
};
```

Slightly Harder Heavy-Light Decomposition

Given a tree consisting of N nodes, answer Q queries of the form:

1. Increase the value of all nodes along the path from a to b by v

2. Change the value of all nodes along the path from a to b to v

3. Find the maximum value of all nodes along the path between two nodes a, b

Slightly Harder Heavy-Light Decomposition

Given a tree consisting of N nodes, answer Q queries of the form:

1. Increase the value of all nodes along the path from a to b by v

2. Change the value of all nodes along the path from a to b to v

3. Find the maximum value of all nodes along the path between two nodes a, b

Solution: Lazy Segment trees with heavy-light decomposition

Ok but what about subtree queries

HLD with subtree queries

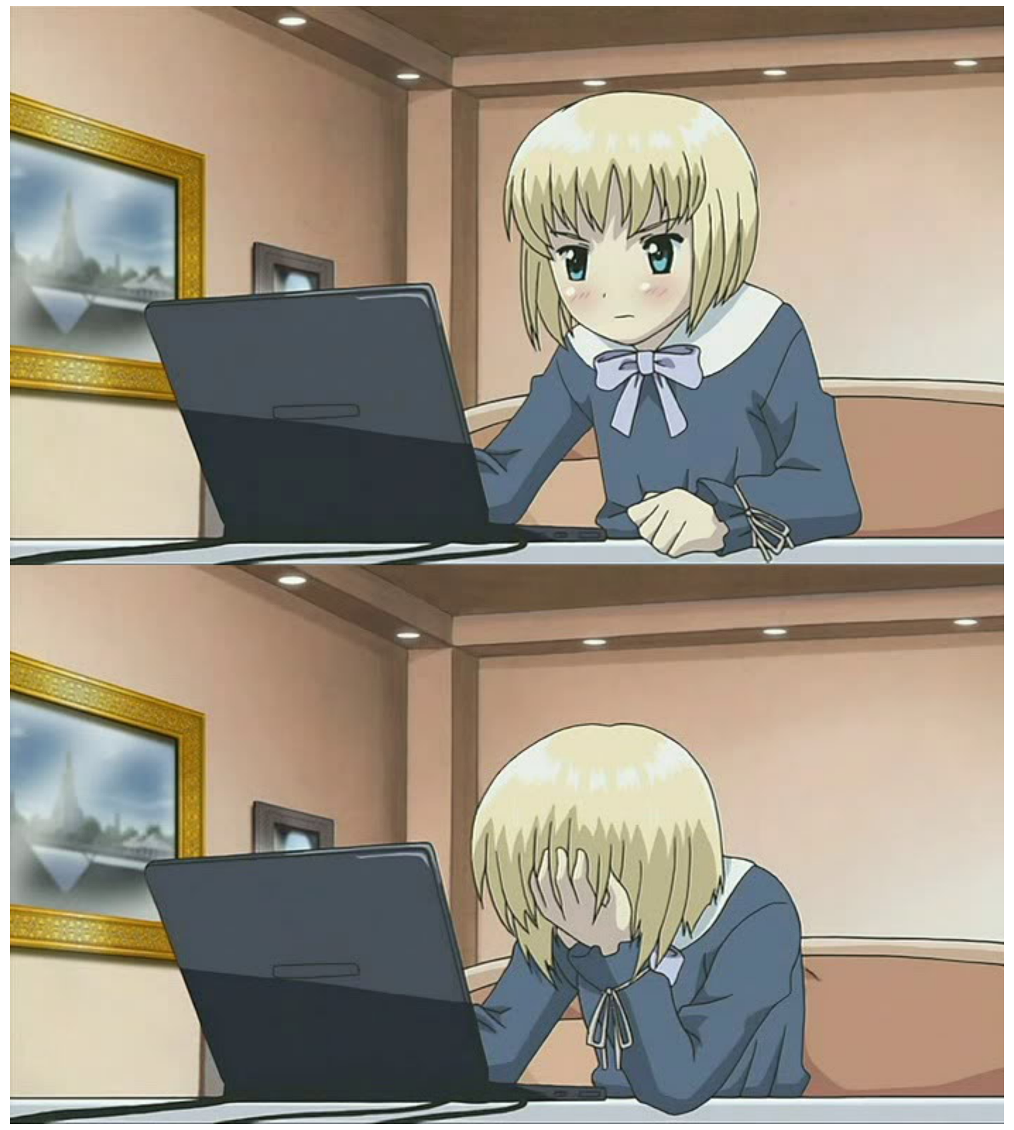
Given a tree consisting of N nodes, answer Q queries of the form:

1. Do something to all nodes along the path from a to b

2. Do something to all nodes in the subtree of s

3. Find some value representing all nodes along the path between two nodes a, b

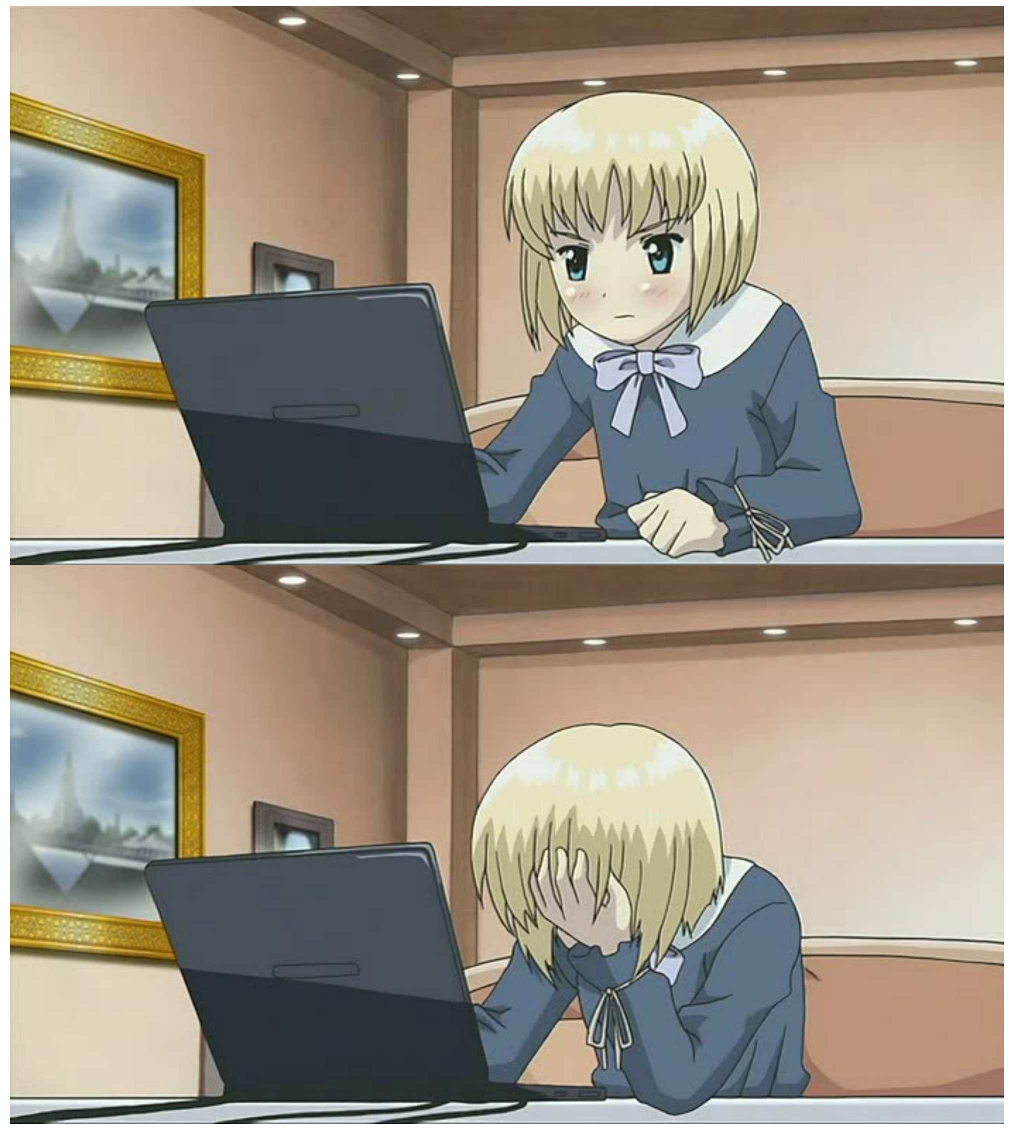
4. Find some value representing all nodes in the subtree of s



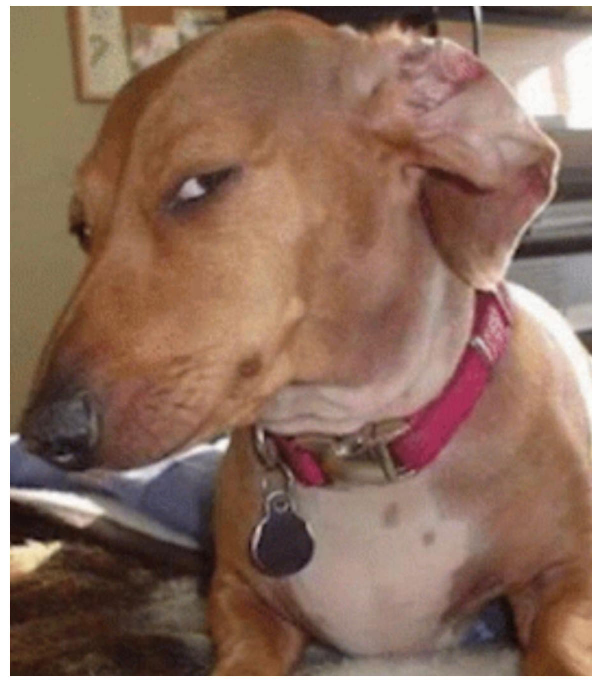
HLD with subtree queries

Given a tree consisting of N nodes, answer Q queries of the form:

1. Do something to all nodes along the path from a to b
2. Do something to all nodes in the subtree of s
3. Find some value representing all nodes along the path between two nodes a, b
4. Find some value representing all nodes in the subtree of s



Solution: ~~Move on to the next problem~~



It can be done!

If we perform an Euler tour of the tree, but always visit the heavy child first, then we still represent heavy paths as continuous parts of the array, and further maintain the property that a subtree is still a continuous part of the array (the whole point of the Euler tour). By recording the exit time of a node we can then query subtrees too! We can combine this with a lazy segment tree to update and query subtrees and paths.

```
void dfs_sz(int u, int p) {
  sz[u]=1;
  for (auto &v : e[u]) {
    if (v==p) continue;
    dep[v]=dep[u]+1; par[v]=u;
    dfs_sz(v, u);
    sz[u]+=sz[v];
    if (sz[v] > sz[e[u][0]]) {
      swap(v, e[u][0]);
    }
  }
}
```

```
void dfs_hld(int u, int p) {
  in[u]=t++;
  a.push_back(v[u]);
  for (auto v : e[u]) {
    if (v==p) continue;
    nxt[v]=(v==e[u][0]?nxt[u]:v);
    dfs_hld(v, u);
  } out[u]=t;
}
```

Example Problems

- CSES 2134: https://cses.fi/problemset/task/2134
- USACO Gold 2019: http://www.usaco.org/index.php?page=viewproblem2&cpid=921
- USACO Gold 2019: http://www.usaco.org/index.php?page=viewproblem2&cpid=970
- USACO Platinum 2018: http://www.usaco.org/index.php?page=viewproblem2&cpid=842
- SPOJ: https://www.spoj.com/problems/QTREE/
- Library Checker: https://judge.yosupo.jp/problem/vertex_set_path_composite
- CF Round 601: https://codeforces.com/contest/1254/problem/D
- JOI 2013: https://oj.uz/problem/view/JOI13_synchronization
- JOI 2018: https://oj.uz/problem/view/JOI18_catdog